VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULACE KŘEHKÝCH TĚLES

BRITTLE BODY SIMULATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ CHLUBNA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/20247/2017/xchlub00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Chlubna Tomáš, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Simulace křehkých těles**
Brittle Body Simulation
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte knihovnu OpenGL a její nadstavby.
2. Seznamte se s metodami simulace pohybu pevných těles (Rigid Body Simulation).
3. Navrhněte způsob tříštění pevného tělesa při nárazu.
4. Popište vybrané techniky a algoritmy.
5. Implementujte navrženou aplikaci.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu.
7. Vytvořte video pro prezentování projektu.

Literatura:

- Podle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4
- Funkční prototyp aplikace

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Milet Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Křehká tělesa se od pevných liší možností roztříštění na menší částečky v souladu se zákony fyziky. Simulace křehkých těles tedy využívá principů obecné simulace pevných těles a vyžaduje vyřešení dalších problémů spojených s rozkladem objektů na částečky a zapojení těchto fragmentů původního objektu do simulace. Popis a zhodnocení možných řešení tohoto problému a návrh s referenční implementací takové simulace jsou cíle této práce.

Abstract

Brittle bodies differ from the rigid ones in the possibility of shattering into small pieces according to the laws of physics. The brittle body simulation therefore uses principles of a general rigid body simulation and requires solutions to the other problems related to object decomposition into fragments and involving these fragments of the original object in the simulation. Description and evaluation of the possible solutions to this problem and a proposal with a reference implementation of such simulation are the goals of this thesis.

Klíčová slova

simulace, fyzika, křehká tělesa, pevná tělesa, renderování, 3D, tříštění, voronoi, detekce kolizí, engine

Keywords

simulation, physics, brittle body, rigid body, rendering, 3D, shattering, voronoi, collision detection, engine

Citace

CHLUBNA, Tomáš. *Simulace křehkých těles.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

Simulace křehkých těles

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Chlubna
19. května 2018

Poděkování

Můj velký dík patří panu Ing. Tomáši Miletovi za trpělivost při vedení, poskytnutí hodnotných informací, zasvěcení do tajů GPU programování a doslova hodiny konzultací. Tímto bych do poděkování rád zahrnul i pana Ing. Michala Kulu za odpovědi na některé otázky z oblasti grafických karet a pana Ing. Michala Španěla (a druhého anonymního autora) za recenze mého článku na konferenci Excel@FIT2018. Dále bych rád poděkoval Fakultě informačních technologií VUT v Brně za znalosti, kterých se mi zde dostalo včetně pana Zdeňka Juříčka za udržování psychického zdraví všech studentů. Děkuji také všem citovaným autorům za zveřejněné informace na dané téma. V neposlední řadě děkuji Ing. Lukáši Bobalíkovi za zajímavé postřehy a mamince za poskytnutí zázemí a morální podporu.

Obsah

Kapitola 1

Úvod

Zatímco simulace pevných těles pokrývá řešení pohybu objektů ve scéně a detekci kolizí, simulace křehkých těles přidává navíc možnost tříštění těles na menší kousky na základě fyzikálních interakcí objektů. Hlavním problémem, který je nutné vyřešit při simulaci křehkých těles ve 3D je rozklad objektu na částečky a následné začlenění těchto částeček do simulace jako nových, rekurzivně dělitelných objektů. Toto dělení musí probíhat pro uživatele přirozeně se jevícím způsobem, podle platných fyzikálních zákonů v aplikaci, běžící v reálném čase. Tato problematika tedy mimo samotné dělení objektu zahrnuje také nutné optimalizační metody pro plynulý běh výsledné aplikace.

Aplikace grafického charakteru (hry, simulace, 3D editory, nástroje pro vizuální efekty apod.) využívají stále více paralelních výpočtů na GPU, proto se i tato práce zaměřuje na efektivní implementaci dané problematiky s využitím grafické karty. Hlavním přínosem práce je popis konstrukce Voroného diagramu a následného štěpení modelu s ohledem na architekturu GPU a s tím spojenou efektivitu výpočtů. Dále jsou popsány vhodné algoritmy zajišťující běh celé simulace a zasahující do oblasti simulací pevných těles.

V teoretické části začínající kapitolou 2 je uveden výčet základních pojmů a návrhových vzorů pro programování GPU. V navazující části 3 jsou popsána křehká tělesa, jejich vlastnosti a význam Voroného diagramu při jejich simulaci. Dále jsou zde popsány také principy sestrojení Voroného diagramu. V sekci 4 se nachází popis algoritmů, zajišťujících pohyb objektů ve scéně. S tím souvisí metody detekce kolizí mezi objekty a následná aplikace fyzikálních zákonů, zajišťující přirozeně se jevící chování objektů ve scéně. Následuje kapitola popisující návrh řešení, kde jsou uvedeny konkrétní vybrané algoritmy, které dohromady tvoří simulaci křehkých těles. Je zde uveden postup sestavení akcelerační struktury pro detekci kolizí v sekci 5.2, výpočtu samotné kolize a reakce na ni v navazující sekci 5.5, popis generování náhodných bodů a box cutting algoritmu pro sestavení Voroného diagramu v sekci 5.3, načež navazuje sekce 5.4 zabývající se aplikací Voroného diagramu při rozdělení modelu. V sekci 6.1 implementační části je popsána struktura celé aplikace a v sekci 6.3 výčet některých konkrétních postupů, použitých při implementaci. Před závěrečným shrnutím a zhodnocením se nachází kapitola 7 popisující výslednou výkonnost aplikace, na základě měření.

Kapitola 2

Simulace na GPU

Algoritmy pro běh simulace je nutno upravit tak, aby byly optimální pro architekturu GPU, zejména z hlediska paralelismu a práce s pamětí. V této kapitole jsou nastíněny základní postupy programování GPU a pojmy, jejichž pochopení je klíčové pro další kapitoly.

2.1 Role GPU

Primárním cílem této práce je aplikace simulačního charakteru s grafickým výstupem. Pro vykreslování 3D scény je využito grafické karty. S tím souvisí možná optimalizace, kdy je geometrie scény přesunuta do paměti GPU a optimálně je celá scéna vykreslena jedním voláním vykreslovací funkce. Jelikož dochází k tomuto přesunu z CPU na GPU, je vhodné, aby byla komunikace mezi CPU a GPU co nejmenší. Proto bude v ideálním případě GPU také provádět všechny výpočty spojené s během simulace a grafická karta je tak využita jako tzv. GPGPU (General-purpose computing on graphics processing units). Další výhodou tohoto přístupu je možnost využití masivní paralelizace, kterou GPU nabízí [17]. Důležitým aspektem je správné rozdělení práce pro daný algoritmus. V úvahu je třeba brát následující struktury:

- kernel - program určený k běhu na GPU
- thread (vlákno) - invokace kernelu
- warp/wavefront (terminologie NVIDIA/AMD) - skupina vláken fyzicky paralelně prováděna na jednom SM (streaming multiprocesoru) na GPU
- workgroup (pracovní skupina) - programátorem definovaná skupina vláken určená k paralelnímu běhu na SM (jeden či více warpů)

Ideálně je v úloze detekována sekvenční část (nejčastěji cyklus iterující nad vstupními daty), kterou lze rozdělit na nezávislé či částečně závislé výpočty a tato část je paralelizována. Takových částí algoritmu může být i více. V takovém případě lze navrhnout více kernelů či vybrat tu část, kde se očekává největší počet oddělitelných výpočtů (iterací). Výpočty lze částečně synchronizovat. V rámci pracovní skupiny lze využít sdílené paměti (shared memory) a programových bariér. Globální synchronizace není možná, pouze lze využít atomických operací nad daty v globální paměti, případně synchronizace mezi spuštěním (dispatch) různých kernelů.

2.2 Paměť

Na GPU lze pracovat se třemi hlavními pamětmi. První z nich je paměť globální, která poskytuje prostor v řádu GB, avšak přístupová rychlost je oproti ostatním poměrně nízká. Příčinou je fyzické umístění paměti mimo výpočetní jednotku. Při práci s globální pamětí je vhodné dodržovat zarovnání, které je v určitých případech automaticky vynuceno OpenGL implicitně (vec3 datový typ je fyzicky uložen se zarovnáním vec4). Zrychlení přístupu může přinést také čtení dat umístěných za sebou v paměti, jelikož řadič paměti může sdružovat více čtení do bloků či využívat cache. Následuje paměť lokální, jejíž velikost je v rámci desítek kB. Tato paměť je několikanásobně rychlejší, proto je často využívána jako cache pro čtení z paměti globální. Paměť je fyzicky umístěna blíže multiprocesoru a díky tomu může být v OpenGL využívána také jako tzv. shared, neboli sdílená paměť. V takovém případě mají k této paměti přístup všechna vlákna dané pracovní skupiny a mohou pracovat nad stejnými daty. Může však nastat problém tzv. bankových konfliktů při čtení ze stejných buněk lokální paměti. Dále je možno využít rychlou konstantní paměť, kde jsou uloženy hodnoty pouze pro čtení. Jako poslední se nabízí sada registrů, které jsou přímo v multiprocesoru. Přístup do registrů je nejrychlejší, ale jejich počet je omezený na vlákno a může ovlivnit výkon s nárůstem vláken a pracovních skupin. V případě nedostatku registrů využije GPU mechanismu *register spilling*, kdy jsou registry emulovány pomocí globální paměti.

2.3 Návrhové vzory

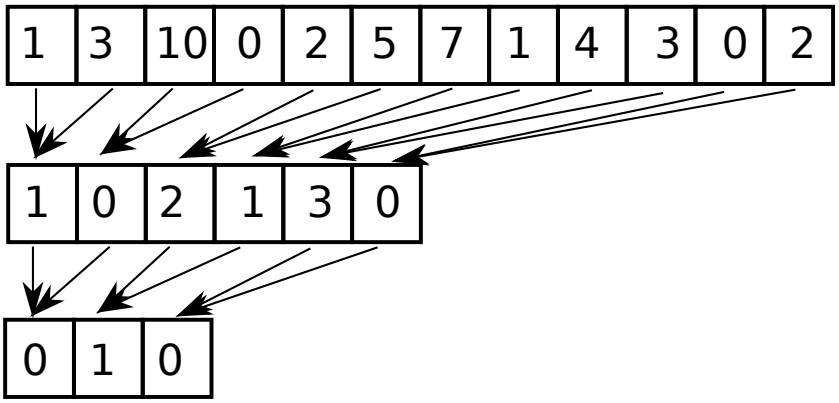
Účelem této práce není popisovat všechny techniky GPU programování, ale ve zbytku textu jsou zmíněny určité metody, které jsou zde stručně popsány.

Paralelní redukce Tento vzor slouží k nalezení určitého elementu z pole dat, nejčastěji maximální či minimální hodnoty (Obrázek 2.1). Namísto iterace v jednom cyklu nad daty jsou data porovnávána po dvojicích (či větších skupinách), kdy vlákno porovná vždy dvě hodnoty, vybere tu lepší a ve druhé úrovni je stejný postup spuštěn nad vybranými hodnotami z první úrovně. Je žádoucí, aby byla vlákna zarovnána těsně za sebou (na posledních úrovních pracuje jeden warp).

Čtení po blocích Náhodný přístup do globální paměti není zcela vhodný, proto může značné zrychlení přinést přeskládání dat tak, aby jeden warp při čtení dat do svých vláken četl ze spojitého bloku v paměti. Řadič paměti pak sdruží požadavky do bloku a může načíst data bez latencí mezi jednotlivými čteními. Případné latence mezi warpy překryjí právě požadavky na paměť více warpů.

Sken a prefixová suma Tento postup lze využít například při řazení, práci s histogramy či eliminaci prázdného místa v řídkých polích. Výsledkem prefixové sumy je pole součtů hodnot daných položek určité části paměti tak, že n-tá položka prefixové sumy je součet hodnot od začátku daného bloku dat do n-té položky těchto původních dat. Například právě při eliminaci mezer v poli lze takto získat nové indexy pro platné položky, přičemž se v prefixové sumě nesčítají hodnoty položek, ale 1 pokud je položka platná a 0 pokud není.

Komprese dat V mnoha případech je nutné ukládat indexy do polí, nahrazující odkazy v programování C/C++. Pro minimalizaci přenosu dat, zejména z globální paměti je žá-



Obrázek 2.1: Paralelní redukce na příkladu hledání minimální hodnoty z pole čísel. Porovnání hodnot probíhá paralelně po dvojicích a výsledek se propaguje postupně na vyšší úrovně. Pro dosažení lepšího výkonu než je na obrázku je výhodné pracovat s první polovinou pole a tvořit dvojice souměrně s hodnotami z poloviny druhé [7].

doucí zmenšit jejich velikost. Například u právě zmíněných indexů většinou plně postačuje poloviční rozsah typu *int*. V GLSL jazyce pro práci s OpenGL compute shadery neexistuje 16bitový datový typ *short*. Komprese dat pak je provedena ručně bitovými posunem a uložením dvou či více hodnot do jedné proměnné. Další možností při například značení aktivních či validních prvků v poli za použití bitových map, kdy každý bit značí stav dané položky přidruženého pole.

Kapitola 3

Křehká tělesa

Křehká tělesa se vyznačují hlavně možností roztříštění na menší kousíčky. K tomuto jevu dochází při dostatečně silné kolizi s jiným objektem či speciálním působení sil, například při explozi. U přesného tříštění těles by bylo nutné studovat krystalickou strukturu atomů a na základě slabších vazeb generovat praskliny v materiálu. Pro potřeby simulace však plně postačí dělení objektu založené na náhodně vygenerovaných bodech v závislosti na druhu materiálu a typu kolize či jiné příčiny tříštění.

3.1 Voroného diagram

Geometrická struktura pojmenovaná po svém objeviteli G. F. Voronoji, také zvaná Dirichletova teselace podle druhého matematika, který ji nezávisle objevil také, je významným modelem používaným v mnoha oblastech vědy. Významnou vlastností této struktury je podobnost s určitými přírodními jevy, které lze pomocí tohoto diagramu dostatečně přesně modelovat.

Definice

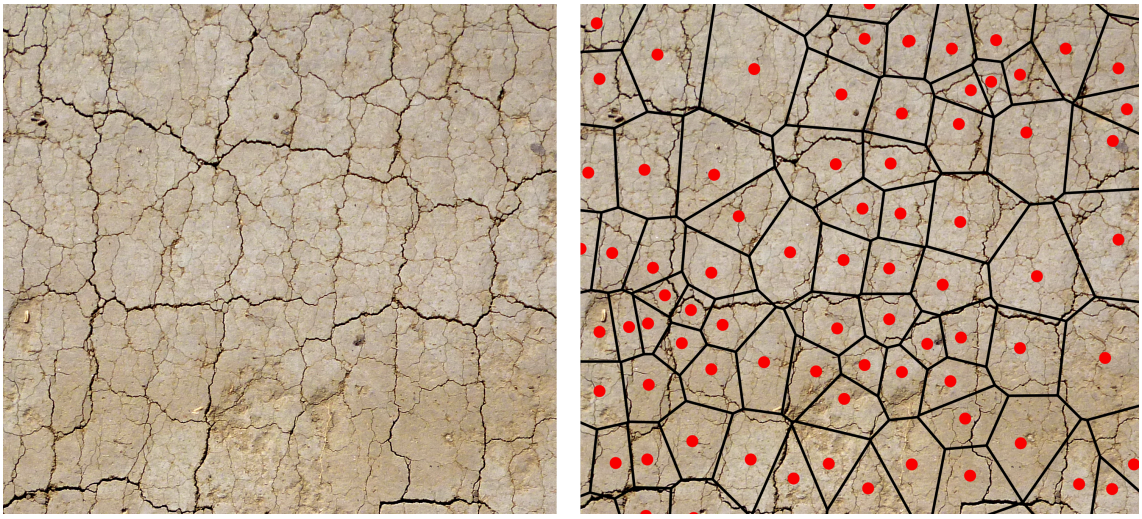
Voroného diagram je definován konečnou množinou bodů (sites). Tyto body mají své prostorové souřadnice a ve výsledném diagramu koresponduje každý bod s jednou tzv. buňkou či oblastí (cell). Na základě vstupních bodů dochází k rozdělení prostoru na buňky tak, že pro každý bod prostoru je nalezen nejbližší bod ze vstupní množiny. Všechny body prostoru, které byly přiřazeny stejnému vstupnímu bodu tvoří buňku. Formálně lze Voroného diagram popsat jako množinu buněk s následující definicí:

$$V(s_i) = \{x \in R^n : |s_i - x| \leq |s_l - x| \forall s_l \in S\} \tag{3.1}$$

kde

- $V(s_i)$ je buňka Voroného diagramu odpovídající vstupnímu bodu s_i
- S je množina vstupních bodů
- R^n je množina bodů n-dimenzionálního prostoru

Takovýto diagram lze sestavit v libovolně rozměrném prostoru. Vzdálenost mezi body v prostoru je nejčastěji definována jako Eukleidovská metrika, v určitých případech, kdy je vyžadován například pravoúhlý pohyb prostorem je také používána metrika Manhattanská. Voroného diagram je znázorněn na obrázku 3.1.



Obrázek 3.1: Voroného diagram sestavený z množiny bodů (červené kroužky). Obrázek demonstruje podobnost vygenerovaného diagramu s přírodními jevy, v tomto případě s prasklinami ve vysušené zemi.

Použití

Vlastnosti Voroného diagramu jsou vhodné pro mnoho různých aplikací spojených se simulací přírodních jevů. Mezi první známé aplikace Voroného diagramu patří studie anglického lékaře Johna Snow, který za pomoci této struktury určil ohniska šíření cholery v Londýně, roku 1854.

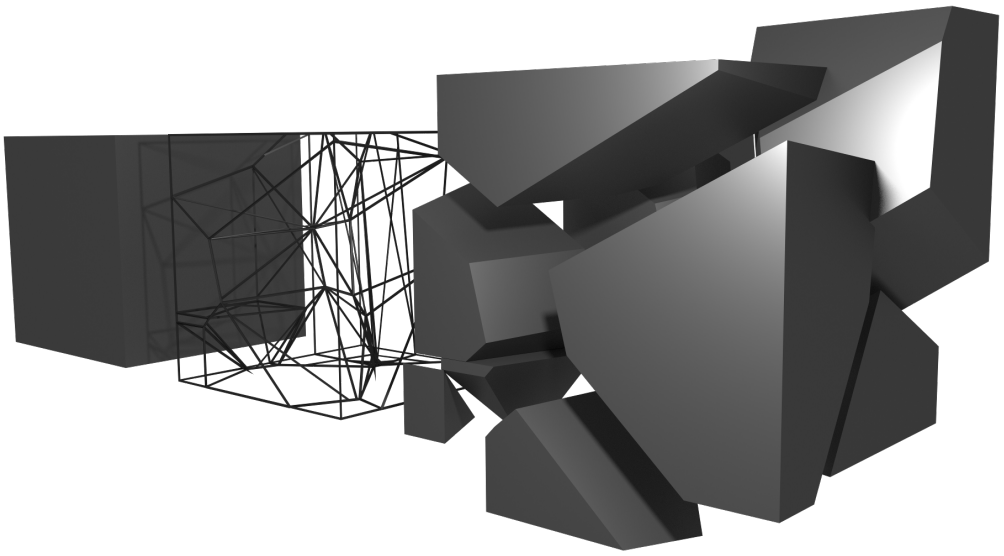
Dále lze tento diagram použít při studiu růstu krystalů, pohybu atomů, v předpovědích počasí (zejména srážek, kde je tato struktura více známá pod názvem Thiessenovy polygony), nebo simulacích socioekonomických jevů [1]. Voroného diagram ve své podstatě simuluje přirozené chování mnoha fyzikálních, biologických a sociálních jevů, jelikož udává rozdělení podle nejbližších bodů a tedy nejméně energeticky náročné cesty, nebo rozložení hmoty či energie.

Pro účely této práce bude tento diagram použit jako simulátor tříštění materiálu. Možnou aplikaci na jednoduchý tvar demonstruje obrázek 3.2. Vlastnosti materiálu lze pak jednoduše měnit pomocí rozložení vstupních bodů. Body mohou být generovány náhodně, takže objekty se budou tříštit pokaždé jiným způsobem. Tak lze i simulovat roztříštění objektu v závislosti na srážce s překážkou, kdy se body vygenerují s větší pravděpodobností okolo kolizního bodu.

Samotnou dekompozici objektů na úlomky lze využít v herním průmyslu, umění, nebo filmových efektech. Dvojrozměrná verze Voroného diagramu má také své využití v těchto oborech, například jako tzv. Voronoi noise (někdy také zván Whorley noise). Tento šum lze využít při efektech energetické povahy (paprsky, exploze, energetické vlny), nebo při procedurálním generování geometrie či textur.

Metody konstrukce

Existuje několik algoritmů, které lze pro konstrukci Voroného diagramu využít. Ne všechny jsou však vhodné pro paralelní zpracování a použití ve 3D prostoru. Pro úplnost je zde uveden stručný seznam často používaných metod.



Obrázek 3.2: Zleva: původní objekt, vygenerovaný 3D Voroného diagram, dekompozice objektu - aplikace předchozích dvou. Voroného buňky lze využít k rozdělení objektu, jelikož struktura těchto buněk věrně připomíná praskliny v reálných objektech. Pozice bodů definují tvary buněk a tak lze simulovat i různé materiály (pravidelná mřížka - kostky/cihly, velké vertikální mezery mezi body - dřevěné třísky atd.)

- **Fortune's algorithm** [9] - Zástupce kategorie tzv. sweep line algoritmů. Celý počáteční prostor se vstupními body postupně zleva prochází vertikální přímka/rovina a postupně přidává body, kterými projde do diagramu. Za ní následuje ve stejném směru tzv. beach line. Tato křivka je sjednocením parabol, vznikajících na levé straně nových bodů a ve výsledku tvořících okraje nových buněk v místech střetnutí dvou parabol.

- **Lloyd's algorithm** [18] - Algoritmus inspirovaný k-means clustering metodou. Iteračně spočítá shluky bodů okolo daných centroidů, provede výpočet nových centroidů pro dané shluky a zpřesní tak výsledek. Je zde však třeba vzorkování prostoru buďto uniformní mřížkou a následný výpočet průměru či za pomoci metody Monte Carlo a náhodně vygenerovaných vzorků. Jakmile jsou nové centroidy dostatečně blízko vstupním bodům, algoritmus končí.

- **Brute force přístup** - Díky možnosti masivní paralelizace na GPU lze také využít voxelizace scény a naivního přiřazení každého voxelu k nejbližšímu vstupnímu bodu diagramu. Tak lze získat diskrétní Voroného diagram. Následně lze využít například marching cubes algoritmu či výpočtů gradientů k získání rovin ohraničujících dané buňky.

- **Extrakce z Delaunayho triangulace/tetrahedronizace** - Voroného diagram lze přímo extrahovat z Delaunayho triangulace ve 2D či tetrahedronizace ve 3D. O možném algoritmu sestavení takové tetrahedronizace pojednávají další kapitoly.

3.2 Dealunayho tetrahedronizace

Jelikož je cílem této práce 3D aplikace, budou se následující kapitoly zabývat právě trojrozměrným Voroného diagramem a tomu náležícími metodami. Voroného diagram lze extrahovat z Delaunayho tetrahedronizace, jejíž konstrukce je popsána mnoha algoritmy.

Definice

Delaunayho tetrahedronizace má stejné vlastnosti jako Delaunayho triangulace ve 2D, jen namísto trojúhelníků figurují v definici tetrahedrony, tedy čtyřstěny, složené z trojúhelníkových stěn. Tetrahedrony jsou vzájemně spojeny tak, že nevznikají díry a celá struktura tvoří konvexní polytop. Průsečíkem dvou takových tetrahedronů může být hrana, bod, stěna či prázdná množina. Tetrahedrony se tedy nepřekrývají. Od obecné tetrahedronizace se liší dodržením tzv. Delaunayho kritéria. Toto kritérium říká, že uvnitř opsané koule daného tetrahedronu nesmí ležet žádný bod jiného tetrahedronu, přičemž na jejím povrchu ležet může. Formálně lze tuto strukturu definovat následovně:

$$DT = \{t \in T : delaunay(t, r) = true, \forall t, r \in T\} \tag{3.2}$$

kde

- T je množina všech tetrahedronů definovaných vrcholy ze vstupní množiny bodů Voroného diagramu
- $delaunay(x, y)$ funkce ověřující splnění Delaunayho kritéria

Dualita s Voroného diagramem

Pokud je k dispozici správně sestavená Delaunayho tetrahedronizace, zkonstruovaná pomocí Voroného vstupních bodů, lze takovou strukturu transformovat právě na Voroného diagram. Obrázek 3.3 demonstruje princip duality ve 2D. Mezi oběma strukturami lze přecházet pomocí aplikace pravidel v tabulce 3.1.

Metody konstrukce

Zde jsou stručně shrnuty hlavní principy několika běžně používaných algoritmů pro sestavení Delaunayovy tetrahedronizace, Jedná se pouze o popis základních myšlenek, jelikož většina těchto postupů není primárně navržena pro běh na GPU.

- **Bowyer-Watson** [16] - Jedná se o nejjednodušší inkrementální algoritmus pro sestavení Delaunayho triangulace v jakékoliv dimenzi. V každém kroku je vkládán jeden bod ze vstupní množiny a tak vznikají nové trojúhelníky spojením s okolními body. V případě že některé trojúhelníky obsahují ve své opsané kouli tento nový bod, jsou smazány a díra takto vzniklá je retriangulována pomocí nově vloženého bodu.
- **Fortune's algorithm** - Pracuje na stejném principu, popsaném v předchozí sekci 3.1.
- **DeWall** [5] - Prostor se vstupními body je rozdělen na dvě části, pro každou z nich je sestavena Delaunay triangulace a následně jsou tyto části spojeny tak, aby nově vzniklé trojúhelníky splňovaly Delaunayho kritérium. Tento postup je prováděn rekurzivně.

Tabulka 3.1: Pravidla převodu Delaunay/Voronoi

Delaunay	**Voronoi**	**Popis**
Bod	Buňka	Množina Delaunayho bodů je stejná jako množina vstupních bodů Voroného diagramu
Hrana	Rovina	Úsečka spojující dva body tetrahedornizace je kolmá na rovinu (polygon) oddělující Voroného buňky
Trojúhelník	Hrana	Voroného hrana je úsečka kolmá na trojúhelník Delaunayho tetrahedronu a vzniká také jako průsečnice dvou Voroného rovin
Tetrahedron	Bod	Bod Voroného diagramu je středem opsané koule daného tetrahedronu

- **Flip-based** [10] - Inkrementální algoritmy, založené na lokálních editacích trojúhelníků pomocí tzv. flipů. O jednom z těchto algoritmů pojednává další kapitola.

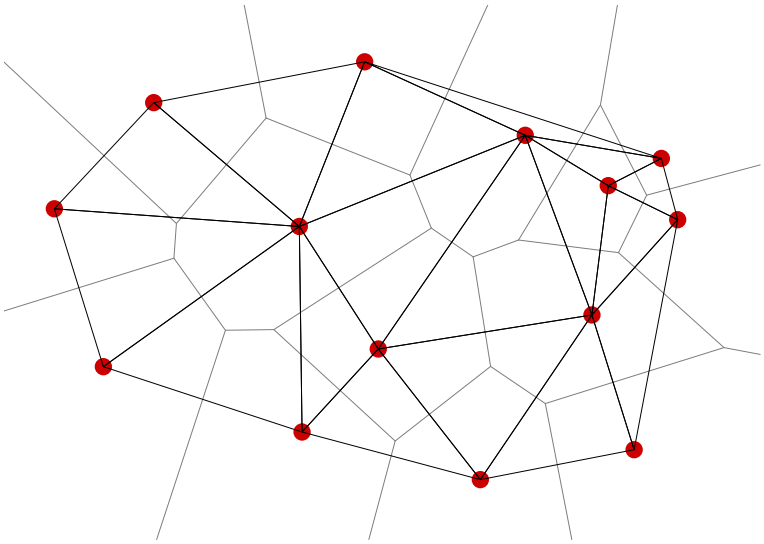
Princip paralelních flip-based algoritmů

Následuje popis základních pojmů a postupů sestavení Delaunayho tetrahedronizace pomocí flip-based metod. Tyto metody lze implementovat částečně paralelně na GPU, avšak stále je patrný inkrementální základ původního algoritmu [10]. Po částech je však možná efektivní paralelizace s nutností řešit možné paměťové konflikty a synchronizace vláken [14].

Základní myšlenkou je postupné vkládání vstupních bodů, kdy výchozí stav je jeden velký tetrahedron prostorově pokrývající celou množinu vstupních bodů. Každý vstupní bod spustí tzv. 1-4 flip. V tomto flipu dojde k rozdělení tetrahedronu na čtyři nové tak, že s nově vloženým bodem jsou spojeny všechny body tetrahedronu, ve kterém tento bod leží. Tento tetrahedron lze identifikovat tzv. walking algoritmem, kdy jsou postupně procházeny tetrahedrony a hledá se takový, kde vstupní bod je orientován uvnitř všech jeho stran. Efektivnější postup je uložení identifikátoru daného tetrahedronu ve struktuře bodu a redistribuce vstupních bodů po každém flipu. 1-4 flip je znázorněn na obrázku 3.4.

Paralelizmu lze dosáhnout postupně s narůstajícím počtem tetrahedronů, kdy každé vlákno provádí výpočet dělení každého tetrahedronu. Na počátku tedy počítá jedno vlákno, ve druhé iteraci čtyři a dále počet vláken narůstá o počet aktuálně dělených tetrahedronů vynásobený čtyřmi. Tyto operace nepotřebují synchronizaci, jelikož jsou čistě lokální.

Aby bylo splněno Delaunayho kritérium, je třeba provést úpravu vzniklé tetrahedronizace pomocí 2-3, resp. 3-2 flipů, vykreslených v obrázku 3.4. Ideálně by tato úprava měla být aplikována po každé iteraci vkládání [4]. V této fázi je provedena kontrola Delaunayho kritéria pro všechny tetrahedrony a jsou označeny ty, které je nutné upravit. Pokud dochází ke konfliktu a jeden tetrahedron je vyžadován pro více flipů, je například pomocí atomických instrukcí deterministicky zvolen tetrahedron s menším indexem. Aby bylo možné provést



Obrázek 3.3: Dualita Voroného diagramu a Delaunayho triangulace (triangulace tvoří spojnice mezi vstupními body)

flip 2-3, je nutné ověřit, zda bude výsledný tetrahedron konvexní. Podobně je nutná kontrola před 3-2 flipem.

Aplikace flipů však nutně nemusí zajistit splnění Delaunayho kritéria u všech tetrahedronů. Při dělení tetrahedronů může dojít ke vzniku neflipovatelné konfigurace a je nutné nakonec aplikovat dodatečné algoritmy pro retriangulaci některých částí. Principem je nalézt vrcholy spojené s problematickými částmi a provést změnu hran propojených s takovými vrcholy.

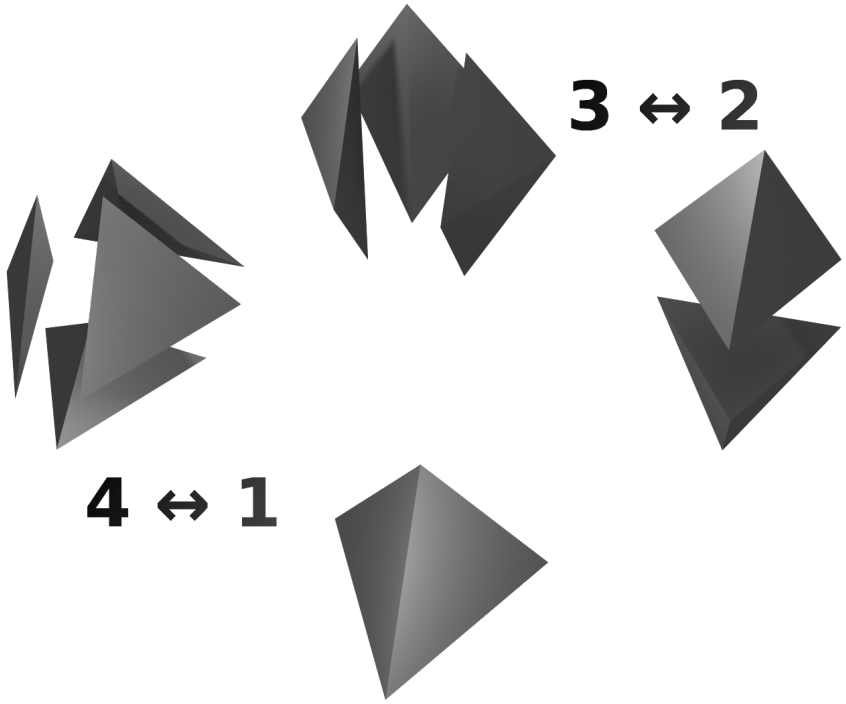
U tohoto postupu je nutné dodržovat vhodně zvolenou konvenci číslování a orientace stěn tetrahedronů a indexaci sousedských vztahů mezi tetrahedrony. Dále je nutné předem alokovat dostatek paměti na GPU a při flipech vhodně zaměňovat původní tetrahedrony za nové a zbývající vkládat na konec seznamu.

Pro rychlé testy orientací bodů je vhodné využít predikátů *orient* a *inSphere* [10]. U obou predikátů lze pouze vypočítat determinant dané matice a v případě že je výsledek kladný, je bod umístěn na kladné straně normály daného trojúhelníka u predikátu *orient* případně uvnitř obalové koule tetrahedronu u predikátu *inSphere*. Obě matice těchto predikátů jsou uvedeny zde:

$$orient(a,b,c,p) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ p_x & p_y & p_z & 1 \end{vmatrix} \quad inSphere(a,b,c,d,p) = \begin{vmatrix} a_x & a_y & a_z & a_x^2+a_y^2+a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2+b_y^2+b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2+c_y^2+c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2+d_y^2+d_z^2 & 1 \\ p_x & p_y & p_z & p_x^2+p_y^2+p_z^2 & 1 \end{vmatrix} \tag{3.3}$$

kde

- $orient(a,b,c,p)$ vrací orientaci bodu p, vzhledem k trojúhelníku a,b,c tak, že kladná strana trojúhelníka je na straně pozorovatele při definici trojúhelníka ve směru hodinových ručiček
  - determinant < 0 - bod je na záporné straně trojúhelníka



Obrázek 3.4: Zobrazení 3D bistelárních flipů 1-4 a 2-3

  - determinant = 0 - bod je na trojúhelníku
  - determinant > 0 - bod je na kladné straně trojúhelníka

- $inSphere(a, b, c, d, p)$ vrací orientaci bodu p, vzhledem k opsané kouli tetrahedronu a, b, c, d definované tak, že $orient(a, b, c, d) > 0$
  - determinant < 0 - bod je vně koule
  - determinant = 0 - bod je na kouli
  - determinant > 0 - bod je uvnitř koule

Kapitola 4

Fyzika

Objekty, ať už se jedná o úlomky po tříštění či o původní tělesa, se musí chovat přirozeně pro lidského pozorovatele. Toho lze nejlépe dosáhnout aplikací fyzikálních zákonů, platných pro reálný svět. Zatímco teoretické výpočty jsou relativně jednoduše realizovatelné, v praxi je nutné zohlednit rozdíly mezi reálným spojitým světem a diskrétním světem počítačové simulace. Následující kapitola nabízí několik postupů, které umožní vytvoření homomorfismu mezi reálným světem a modelem vhodným pro běh v počítačové simulaci.

4.1 Spojitý a diskrétní čas

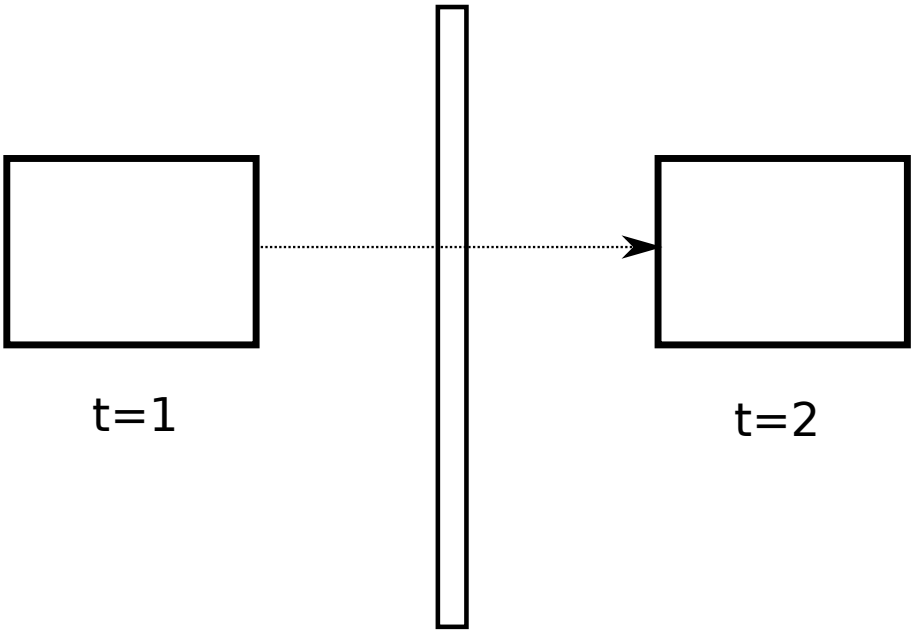
Diskretizace spojitých hodnot vždy přináší určitá omezení a možnost výskytu chyb při výpočtech. U simulací je tento problém zvlášť výrazný, jelikož je nutné spojitost nějakým způsobem napodobit. Nejčastěji jde o splnění určitých podmínek pro spuštění dané akce. Simulátor kontroluje, zda daná veličina dosáhla určité hodnoty. Jelikož však k nárůstu její hodnoty dochází skokově, je možné, že očekávaná hodnota bude přeskočena a bude možné detekovat splnění podmínky až po překročení dané hranice.

K řešení tohoto problému lze využít například metodu půlení intervalů, kdy je nutné upravit krok definující rozdíl mezi původní hodnotou veličiny a novou hodnotou. K ukládání možných akcí v určitých časech lze využít kalendáře událostí, množiny dvojic akce/aktivační čas. Řeší se tak tzv. kombinované (diskrétní a spojité) simulace. [15]

V případě fyzikálních simulací dochází k tzv tunelování, kdy objekt při své skokové změně polohy v čase, simulující pohyb tělesa, může přeskočit překážku. K tomuto jevu dochází v případě pohybu objektů velkou rychlostí či pohybu objektu skrze tenkou překážku. Princip tunelování demonstruje obrázek 4.1. Možným řešením je aplikace metody detekce kolizí s možností predikce na základě rychlosti objektu.

4.2 Detekce kolizí

Aby bylo možné vůbec realizovat interakce objektů, mezi sebou navzájem ve scéně, je nutné zjistit zda dva či více objektů mezi sebou v daném čase kolidují. Při řešení tohoto problému je nutné vycházet z reprezentace objektů v dané aplikaci. V případě této práce se jedná o modely definované trojrozměrnou triangulací, případně doplněné o dodatečné atributy a fyzikální vlastnosti. Následující kapitoly popisují dva hlavní kroky detekce kolizí používané ve fyzikálních simulacích.



Obrázek 4.1: Tunelování - problém diskrétního času v simulaci fyziky (objekt ve dvou časech s překážkou, skrze kterou prošel ve své trajektorii)

Broad-phase

Scéna může obsahovat velké množství objektů a je žádoucí rozhodnout, které dvojice objektů budou spolu testovány. Vstupem broad-phase algoritmů je množina všech objektů dané scény a výstupem jsou její podmnožiny obsahující objekty, které mají vysokou pravděpodobnost vzájemné kolize. K tomuto účelu lze využít několika možných metod. Mezi nejefektivnější patří metody založené na dělení prostoru. Popis často používaných postupů shrnuje tento seznam:

- **Brute force** - Nejjednodušší způsob výběru objektů je vytvoření dvojic každého objektu s každým s eliminací duplikátů a testování objektu se sebou samotným. Tento postup je vhodný pro velmi malé scény s jednotkami či maximálně několika desítkami objektů.
- **Obalová tělesa** - Také zvaná *bounding volumes* jsou zjednodušené modely, obalující složitou geometrii a tak redukující výpočetní náročnost kolizních testů. Mezi často používaná tělesa patří BS (bounding spheres), AABB (Axis-aligned bounding boxes), OBB (Oriented bounding boxes), slaps a další relativně jednoduché tvary (Obrázek 4.2). V broad-phase je možné na základě kolizí těchto tvarů určit potenciální kolize objektů.
- **Binární dělení prostoru** - Scénu lze rekurzivně dělit na dvě poloviny, čímž vznikne binární strom, kde každý uzel reprezentuje množinu prostorově blízkých objektů. Uzly poblíž listových lze pak použít k identifikaci potenciálně kolizních skupin.
- **Bounding Volume Hierarchy** - Lze také implementovat jako binární strom, kde však existuje více možných postupů sestavení. Hlavní myšlenkou je opět seskupit

blízké objekty do jednotlivých podstromů. Jedním z možných postupů je například seřazení objektů pomocí prostor vyplňující křivky a následné rozčlenění této posloupnosti na uzly stromu. Vztah mezi scénou a stromovou strukturou demonstruje obrázek 4.3.

- **Octree** - V případě octree dochází k pevně danému rekurzivnímu dělení prostoru scény na osm buněk (čtyři v případě dvojrozměrného quadtree). Celá scéna je většinou obalena do jedné velké krychle, která je dále dělena jako na obrázku 4.4, dokud není dosažena limitní úroveň dělení či dokud nejmenší buňky neobsahují limitní počet objektů. Strukturu reprezentuje opět strom s osmi potomky pro každý rodičovský uzel. Dělení tedy nastává jen v případě potřeby, když vkládaný objekt nelze vložit do volného listového uzlu v daném podstromu.

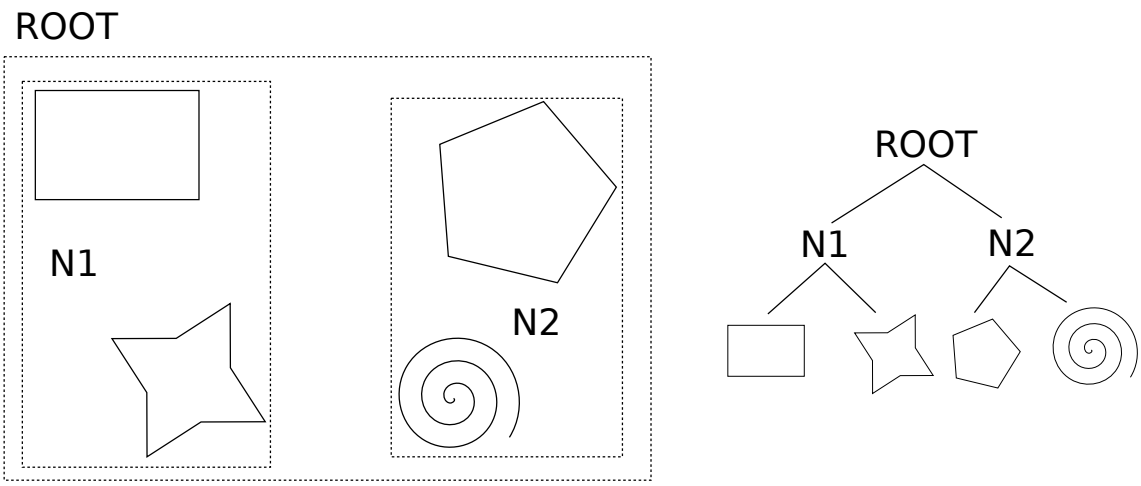
- **Uniformní mřížka** - Podobný princip jako u octree, postrádající adaptivitu na rozložení objektů ve scéně. Scéna je pravidelně rozdělena na buňky, sdružující blízké objekty.

Obrázek 4.2: Obalový válec, koule a krychle. Obalová tělesa musí splňovat tři atributy. Zaprvé musí co nejtěsněji přiléhat k povrchu obalované geometrie, zadruhé musí tvarem co nejlépe odpovídat obalované geometrii a zatřetí musí být výpočet jejich vzájemné kolize optimálně implementovatelný.

Narrow-phase

Druhá fáze detekce kolizí má za úkol rozhodnout, zda ke kolizi objektu dochází na základě skutečné geometrie modelů. Dále je často žádoucí získat dodatečné informace o místě kolize či míře vzájemného průniku modelů. Tato fáze je obecně výpočetně náročná, proto jí předchází redukce testovaných dvojic pomocí broad-phase metod. Většina algoritmů je navržena pro konvexní tvary. Objekty jsou většinou předzpracovány a rozděleny na konvexní části pro další výpočty. Opět je více možností, jak rozhodnout problém kolizí u obecných triangulovaných modelů.

- **Per triangle** - Naivním postupem může být průchod trojúhelníky obou modelů a testování na vzájemné kolize pro všechny dvojice. Tento test lze provést hledáním průsečnice dvou rovin a ověřením, zda leží obou trojúhelnících zároveň. [13] Tento test lze optimalizovat pomocí dekompozice modelů na vhodné akcelerační struktury.

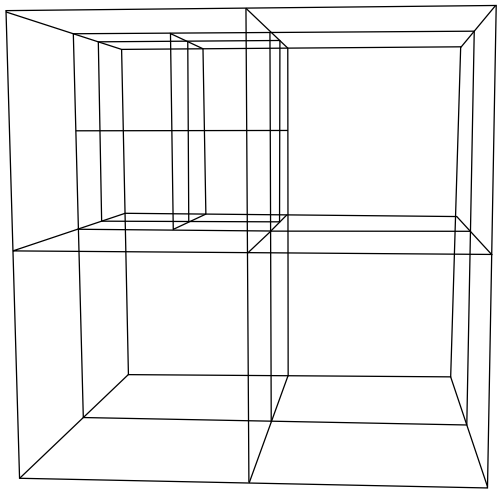


Obrázek 4.3: Bounding volume hierarchy nad scénou ve 2D, reprezentována jako binární strom. Objekty jsou postupně zapouzdřeny do několika úrovní obalových těles podle dané funkce vzájemnné blízkosti.

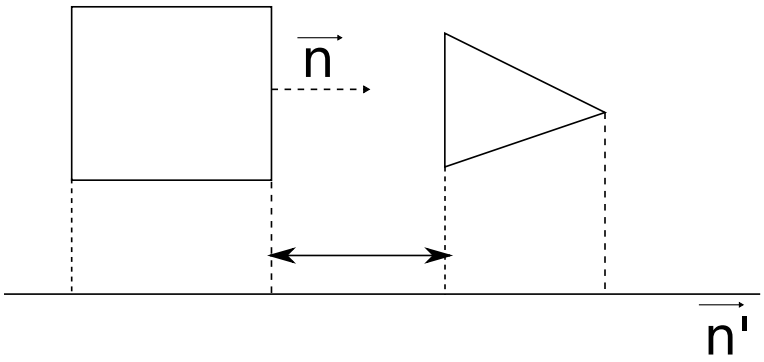
- **Separating Axis Theorem** - tento postup využívá projekce vrcholů modelu na obecnou osu v prostoru. Tak je získán interval, který daný model na této ose zabírá a následně je možné zjistit, zda dva objektu spolu na této ose mají či nemají kolizi. Pokud existuje osa, kde dva objekty kolizi nemají, kolize nenastává. Tato osa je nazývána separační osou [6]. Jelikož nelze testovat všechny možné osy, kterých je v prostoru nekonečně mnoho, probíhají testy na všech normálách obou objektů, které jsou dostačující jelikož definují všechny možné situace kolize (pro přesnější výpočty jsou někdy také přidávány osy kolmé k hranám či rohům). Nalezení separační osy znázorňuje obrázek 4.5.

- **Gilbert–Johnson–Keerthi distance algorithm** - Algoritmus určený k zjištění vzdálenosti mezi dvěma objekty. Tento algoritmus však může také určit výskyt kolize. [2] K tomuto účelu lze použít Minkowskeho sumy, která definuje nový konvexní tvar vytvořený součtem všech vrcholů dvojice objektů. Namísto součtu však lze vrcholy od sebe odečítat a získat tak nové souřadnice. Pokud takto získaný tvar obsahuje počátek souřadného systému, dochází ke kolizi původních modelů. Aby nebylo nutné provádět rozdíly všech souřadnic obou modelů, lze algoritmus implementovat iterativně a testovat vždy minimální počet bodů tvořících simplex dané dimenze. Body pro rozdíl jsou vybrány vždy jako nejvzdálenější body v daném směru (u druhého tvaru ve směru opačném). Jako další směr se vždy vybírá normála ke straně nejblíže počátku souřadnic ve směru tohoto počátku. Cílem je vždy počátek překročit a zahrnout jej do výsledného tvaru. Pokud v nově určeném směru žádný další bod neleží, lze algoritmus ukončit bez nalezení kolize.

- **Dekompozice na obalová tělesa** - Modely lze aproximovat pomocí obalových těles. Jedno obalové těleso samozřejmě nepokryje vhodně obecný tvar modelu, avšak je možné použít více takových těles a rozdělit tak objekt na jejich soustavu. Přesnost takové detekce závisí na výběru obalových tvarů a na počtu takových tvarů aplikovaných na model. Obecně tato metoda nezaručuje maximálně přesnou detekci.



Obrázek 4.4: Prostor, ohraničený krychlí je podle potřeby (výskytu objektů) rekurzivně dělen na 8 menších krychlí. Scéna je pak reprezentována jako octree, strom kde každý vnitřní uzel má právě 8 potomků.

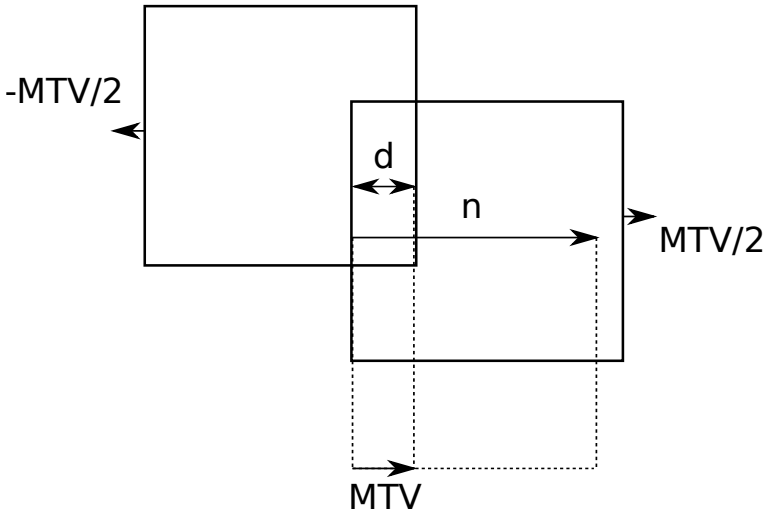


Obrázek 4.5: Separační osa jako jedna z normál u SAT. Oba objekty jsou na tuto osu projektovány a je hledán případný průnik těchto projekcí. Pokud je naleyena osa bez průniku, je označena jako separační a tak je zjištěno že ke kolizi dvou objektů nedochází.

Vzhledem k obtížné predikci kolizí, způsobených rotačním pohybem těles, nebo v případě vícenásobných kolizí může dojít k průniku dvou geometrií. Detekční algoritmy narrow-phase většinou vrací Minimum Translation Vector (MTV) udávající kolizní normálu a míru překrytí objektů ve směru této normály. Objekty pak lze posunout, každý o polovinu MTV ven z kolize podle normály jako na obrázku 4.6.

4.3 Chování objektů po kolizi

Po nárazu obvykle dojde k přenosu energie mezi objekty a vlivem fyzikálních vlastností objektů dojde silovým působením ke změně rychlostí. Fyzikální jevy je nutné aproximovat zjednodušenými výpočty nad ideálními tělesy pomocí fyzikálních zákonů.



Obrázek 4.6: Vzájemné vysunutí objektů z kolize o polovinu MTV (d označuje míru vzájemného průniku objektů a n kolizní normálu).

Rázy a moment síly

Výsledný stav objektů lze popsat jednoduchými fyzikálními vzorci popisujícími tzv. rázy:

$$v_a = \frac{m_a - km_b}{m_a + m_b} v_a + \frac{(1+k)m_b}{m_a + m_b} v_b, v_b = \frac{(1+k)m_a}{m_a + m_b} v_a + \frac{m_b - km_a}{m_a + m_b} v_b \tag{4.1}$$

Rozlišujeme dva krajní případy rázů, kdy u obou dochází k zachování hybnosti. V případě že je zachován také zákon zachování mechanické energie, mluvíme o pružném rázu. Pokud je mechanická energie soustavy menší po srážce, jedná se o nepružný ráz (energie je přeměněna na zvuk, teplo, deformaci materiálu, atd.). Míru zachování mechanické energie určuje koeficient restituce. Pokud je nulový tak se jedná o dokonale nepružný ráz, pokud je roven jedné tak o dokonale pružný ráz. Tento koeficient je definován pro dvojice materiálů experimentálně.

Otáčivé účinky lze popsat pomocí momentu síly. Jedná se o vektorovou veličinu, která udává směr otáčení a velikost, tedy míru či rychlost otáčení. Vektor momentu síly a jeho velikost lze vypočítat následovně:

$$M = r \times F, |M| = rFsin(\alpha) \tag{4.2}$$

kde

- F je síla působící na objekt v bodě kolize
- r je vektor s počátkem v bodě osy otáčení a koncem v bodě kolize
- R^n je množina bodů n-dimenzionálního prostoru

Výše zmíněný postup je funkční v případě jednoduchých scén, avšak u vícenásobných kolizí může způsobovat nechtěné exploze, průniky těles a nekonečné odrazy namísto ustálení objektu.

Impuls síly

Vhodným modelem popisujícím pohyb objektů po kolizi je impulzní model, který také umožňuje simulovat tření. Impuls síly je definován jako integrál působící síly v čase. V reálném světě při kolizi na sebe objekty po určitý čas vzájemně silově působí a na základě výsledných sil lze určit nové rychlosti. Takový výpočet by byl velmi komplexní a sahal až na úroveň atomů. Situace v simulacích je většinou zjednodušena tak, že je žádoucí spočítat výsledné rychlosti přímo bez integrací a závislosti na čase.

Ze základních rovnic lze odvodit rovnici pro výpočet výsledného impulsu síly a tento impuls pak jen aplikovat na rychlosti objektů. Tak lze získat nové rychlosti. Podrobné odvození lze nalézt v [8]. Zde jsou uvedeny funkční vztahy pro výpočet rychlostí:

$$r_a = cp - mc_a, r_b = cp - mc_b \tag{4.3}$$

$$v_{ab} = (v_a + (\omega_a \times r_a) - v_b - (\omega_b \times r_b)) \cdot n \tag{4.4}$$

$$i = \frac{-(1+k)v_{ab}}{m_a^{-1} + m_b^{-1} + (I_a^{-1}(r_a \times n) \times r_a + I_b^{-1}(r_b \times n) \times r_b)n} \tag{4.5}$$

$$v_a = v_a + \frac{in}{m_a}, v_b = v_b - \frac{in}{m_b} \tag{4.6}$$

$$\omega_a = \omega_a + r_a \times (in)I_a^{-1}, \omega_b = \omega_b - r_b \times (in)I_b^{-1} \tag{4.7}$$

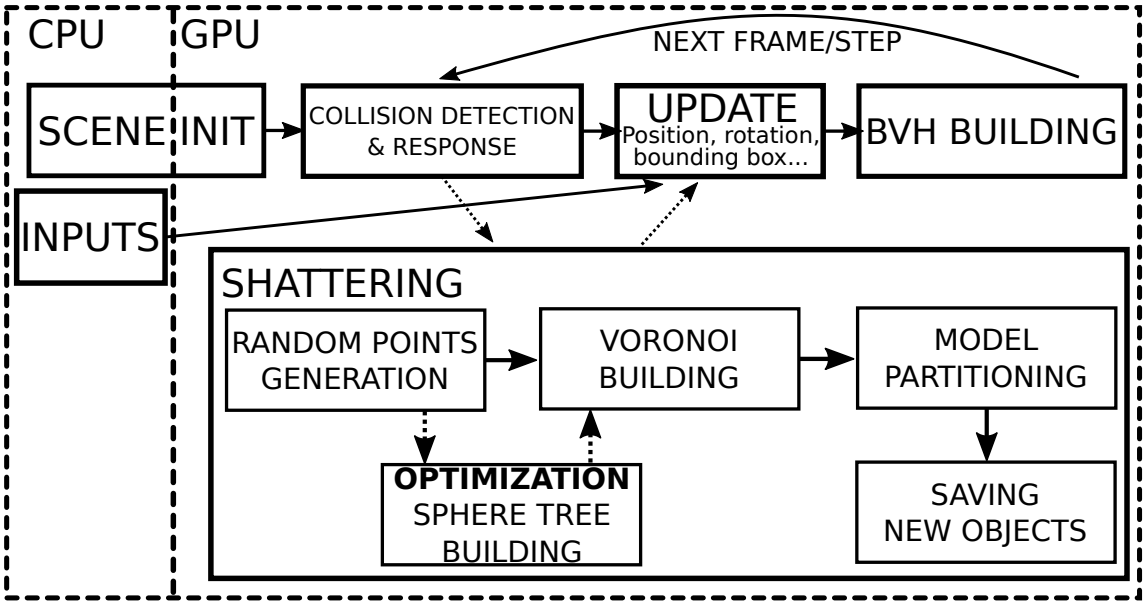
kde

- cp je kolizní bod
- mc je pozice těžiště
- n je normála kolize, směřující podle konvence od objektu a k objektu b
- m je hmotnost objektu
- k je koeficient restituce
- I je moment setrvačnosti, lze použít obecné vzorce pro tvar blížící se tvaru objektu

Kapitola 5

Návrh řešení

V této kapitole je popsán způsob řešení simulace křehkých těles a souhrn použitých technik v této práci. Nachází se zde konkrétní postup rozkladu těles na částečky, detekce kolizí a řešení fyziky. Voroného diagram bude sestaven přímo pomocí testování rovin mezi body a detekce kolizí bude akcelerována pomocí BVH. Celý proces je schématicky naznačen na obrázku 5.1.



Obrázek 5.1: Schéma hlavní smyčky aplikace. CPU nahraje a inicializuje scénu, kterou následně zašle do paměti GPU. Výpočty jsou pak prováděny na GPU, kdy CPU se stará pouze o neustálé opakování hlavní smyčky, volání potřebných knihovních funkcí pro vykreslování a spouštění kernelů a zpracovávání uživatelských vstupů. V případě potřeby je spuštěn kernel pro tříštění objektu.

5.1 Výpočty na GPU

Jak již bylo naznačeno v teoretické části, cílem je implementovat simulaci celou na GPU. Simulace křehkých těles sama o sobě nemá mnoho využití, proto je často nutné ji použít v určitém kontextu. Mezi nejčastěji používané patří filmové efekty a aplikace herního

charakteru. Přesunutím reprezentace scény na GPU je minimalizována komunikace mezi procesorem a grafickou kartou. Dále jsou kolize, fyzika i tříštění počítány na GPU. CPU je tedy zodpovědné pouze za zpracování vstupů. V rámci herního enginu by bylo možné takto uvolnit výkon pro umělou inteligenci a další herní mechanismy, přičemž by mezi GPU a CPU probíhala pouze komunikace ohledně vzniklých změn na jedné straně, relevantních pro stranu druhou. Herní enginy a fyzikální knihovny, jako je například Bullet Physics, již postupně GPU akceleraci náročných výpočtů zavádí.

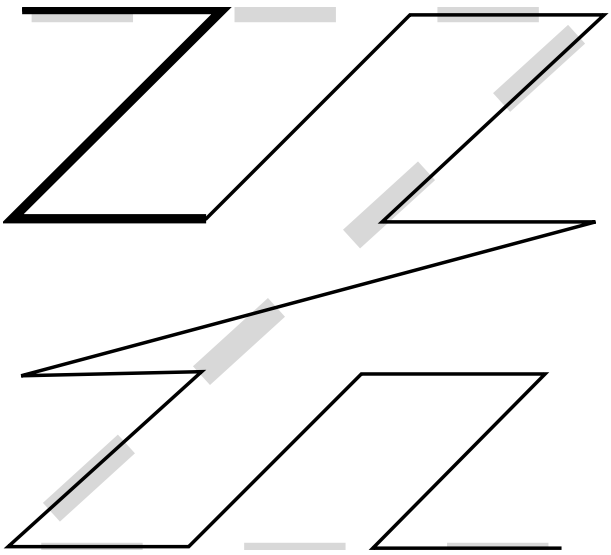
5.2 Bounding volume hierarchy

BVH lze sestavit rychle a paralelně na GPU, proto je tato struktura zvolena pro tuto práci. BVH slouží k výběru potenciálně kolidujících objektů v broad-phase detekci kolizí. Následuje několik pojmů, klíčových pro uvedený postup sestavení BVH.

Mortonův kód

Prostor lze vyplnit rekurzivním průchodem tzv, prostor vyplňujících křivek. Jednou z nich je Z-křivka, pomocí které lze převést trojrozměrnou scénu na lineární posloupnost objektů, přičemž dojde k zachování sousednosti (objekty, které byly umístěny blízko sebe v prostoru jsou blízko i v lineární posloupnosti).

Hlavní výhodou této křivky je možnost efektivního sestavení pomocí Mortonova kódu. Mortonův kód lze získat ze souřadnic objektu prostým proložením binárních reprezentací tří souřadnic objektu. Jelikož je výhodné pracovat s 32 bitovými typy na GPU, je použit 30 bitový Mortonův kód. Průchod této křivky prostorem lze vidět na obrázku 5.2.

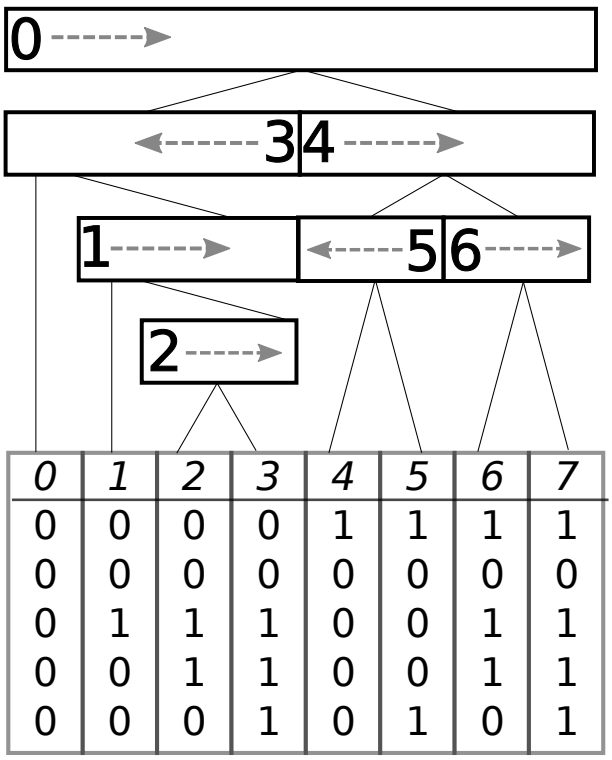


Obrázek 5.2: Z-křivka rekurzivně vyplňující prostor (začátek vlevo nahoře, cestou vytváří trajektorii ve tvaru písmene Z, tyto útvary opět vytvoří tvar Z atd.)

Binární radixový strom

Tento strom, který ve výsledku bude reprezentovat scénu jako BVH, lze sestavit nad seřazenou posloupností například binárních řetězců tak, že každý rodičovský uzel reprezentuje

podstrom, kde uzly tohoto podstromu mají stejný společný prefix definovaný kořenem tohoto podstromu. Ve stromu nejsou ukládány prázdné uzly, ale jen skutečné prefixové uzly či listy obsahující konečné elementy. Tato struktura tedy nezabírá zbytečné místo v paměti. Další výhodnou vlastností je počet rodičovských uzlů, kterých je vždy N-1, přičemž N je počet listů, Ve výsledku je tedy pouze nutné přiřadit každému rodičovskému uzlu část lineární posloupnosti seřazených Mortonových kódů. Výsledný strom, vytvořený dělením lineární posloupnosti je znázorněn na obrázku 5.3, kde je možné si také povšimnout, že každá sekvence prvků v uzlu začíná prvkem se stejným indexem, jako je index uzlu. Směr pokračování sekvence se pak liší podle dělení.



Obrázek 5.3: Binary radix tree sestavený nad seřazenou lineární posloupností Mortonových kódů. (seznam kódů dole, obdelníky nad seznamem znázorňují dělení na uzly stromu kde číslo je počáteční element a šipka směr pokračování uzlu ve vztahu k seznamu všech elementů, úsečky mezi úrovněmi znázorňují dělení uzlů)

Paralelní radix sort

Základní ideou radix sortu je iterativní řazení po jednotlivých číslicích (či jiných částech) dané hodnoty (obecně řetězce). Samotné řazení po jednotlivých elementech probíhá pomocí prefixové sumy. Na začátku každé iterace je spočítán histogram, udávající kolik výskytů daného elementu se na testované pozici v aktuální iteraci nachází. Na histogramu je následně vypočítána prefixová suma tak, že na i-té pozici se nachází vždy součet hodnoty na pozici a i-1, přičemž počáteční hodnota se nemění. Následně jsou od posledního prvku posloupnosti k řazení vybírány prvky a vkládány vždy na pozici, určenou hodnotou v prefixové sumě. Tato hodnota je však nejprve dekrementována o jedna a uložena. Jednoduché řazení ve dvou iteracích je demonstrováno v tabulce 5.1

Tabulka 5.1: Demonstrace řazení pomocí radix sortu

první iterace (první číslice z leva)					
čísla	24	1	40	2	4
histogram	**0**	**1**	**2**	**3**	**4**
	1	1	1	0	2
prefixová suma	1	2	3	3	5
	0	1	2		4
					3
výstup	40	1	2	24	4
druhá iterace (vstupem je výstup z první iterace)					
histogram	**0**	**1**	**2**	**3**	**4**
	3	0	1	0	1
prefixová suma	3	3	4	4	5
	2		3		4
	1		2		
výstup	1	2	4	24	40

Paralelně lze algoritmus optimalizovat. Histogramy je možno počítat lokálně v rámci pracovní skupiny. Každé vlákno analyzuje jeden Mortonův kód, projde pozice jeho číslic a pomocí atomických instrukcí inkrementuje příslušné místo v lokálním histogramu. Počítá se tak najednou histogram pro každou iteraci, tedy histogram výskytu číslice na dané pozici pro všechny pozice. Výsledkem je lokální pole histogramů. Paralelně je pak pomocí atomických instrukcí proveden součet všech histogramů do jednoho globálního pole histogramů. Pro každý histogram je paralelně provedena prefixová suma. V této fázi lze již provést výsledné řazení pomocí prefixových sum.

Souřadnici lze nakvantovat vzhledem k velikosti scény a datové šířce následovně:

$$quantizedCoord = \frac{(coord - sceneMin)}{sceneMax - sceneMin} \cdot 2^{bitCount} \tag{5.1}$$

Algoritmus sestavení

Celý algoritmus sestavení lze tedy popsat následovně [11]:

1. Paralelně spočítat axis-aligned bounding box pro každý objekt - maximum a minimum ze souřadnic vrcholů v každé ze tří souřadných os
2. Paralelně nakvantovat souřadnice středů těchto obalových těles a získat jejich Mortonovy kódy
3. Provést řazení seznamu kódů pomocí paralelního radix sortu
4. Paralelně pro všechny rodičovské uzly s indexem n ($n \in \langle 0, N-2 \rangle$, kde N je počet objektů)
   (a) **Určit směr sekvence** - se kterym sousedem má objekt s indexem n větší společný prefix: $d = sign(commonPrefix(n, n+1) - commonPrefix(n, n-1))$

(b) **Získat délku sekvence v daném směru** - pomocí binárního vyhledávání (možná optimalizace pomocí exponenciálního vyhledávání) nalézt takový prvek ve směru sekvence, jehož společný prefix s n-tým prvkem (začátkem sekvence) je menší než n-d prvek (prvek před začátkem sekvence)

(c) **Nalézt bod dělení uzlu** - opět pomocí binárního hledání, cílem je nalézt nejvzdálenější kód od začátku sekvence v daném směru, jehož společný prefix se začátkem sekvence je větší než společný prefix hledaného a konce sekvence

(d) **Přiřazení potomků danému uzlu** - podle směru bude jeden potomek podsekvence od začátku aktuálního uzlu po nalezený bod dělení a druhý potomek bude podsekvence od kódu za bodem dělení do konce rodičovské sekvence (pokud obsahuje sekvence potomka jen jeden kód, jedná se o list)

5.3 Voroného diagram

Postup sestavení Voroného diagramu za použití Delaunayho tetrahedronizace lze implementovat částečně paralelně. Algoritmy založeny na tomto postupu však vychází z původně sekvenčního inkrementálního algoritmu a tato vlastnost se stále projevuje. Například k paralelizaci tvoření tetrahedronů dochází postupně, je třeba iterativně spouštět více kernelů a je nutné na konci sekvenčně opravit případné neflipovatelné stavy. Proto je v této práci navržen přímý postup sestavení pomocí box cutting algoritmu.

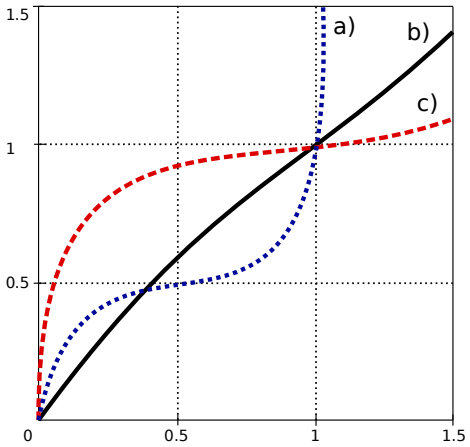
Generování náhodných středů

Vstupem pro algoritmus sestavení Voroného diagramu je množina bodů, které budou tvořit středy Voroného buněk. Generování pseudo-náhodných hodnot na GPU je možné pomocí vzorkování chyby funkce sinus v oblasti desetinných míst (Výpis 6.1). V případě generování bodů bez ohledu na vztah ke geometrii modelu lze jednoduše vygenerovat jednotlivé souřadnice bodů paralelně v rámci obalové krychle modelu. Jelikož budou středy Voroného buněk odpovídat úlomkům, je nutné také upravovat rozložení těchto bodů tak, aby bylo možné vygenerovat více úlomků například okolo bodu kolize dvou těles. K tomu lze využít inverzní transformace funkce rozložení pravděpodobnosti a modelovat výslednou funkci pomocí funkce tangens jak je naznačeno na obrázku 5.4.

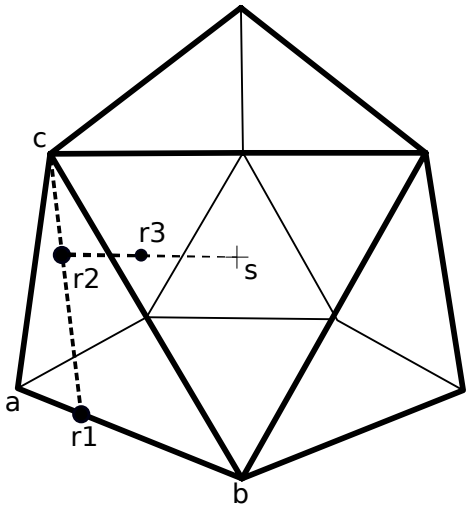
Při sestavování Voroného diagramu v další fázi však jsou středy buněk mimo geometrii modelu nežádoucí a mohou vést k nechtěným stavům jako na obrázku 5.12. Proto je v této práci použita jiná metoda, respektující povrch modelu za cenu menší kvality výsledného rozložení. Toto zhoršení kvality však není patrné při běžném použití v aplikaci simulačního charakteru. Tuto metodu demonstruje obrázek 5.5.

Box cutting algorithm

Každá pracovní skupina na GPU je přiřazena jednomu Voroného středu ze vstupní množiny. Jedna Voroného buňka je tedy počítána na jednom multiprocesoru, což algoritmu umožňuje využít lokální paměti pro uložení dat reprezentujících buňku. Buňka je reprezentována jako množina úseček, odkazujících se do pole vrcholů. Každá úsečka je také asociována se dvěma indexy rovin, jejichž hranu tvoří (uvažujeme pouze manifold objekty, tudíž bude mít každá úsečka přesně dvě roviny). Na počátku je buňka inicializována jako AABB celého modelu. Algoritmus pak vybere nejbližšího souseda pro daný bod ze vstupní množiny středů, nalezne



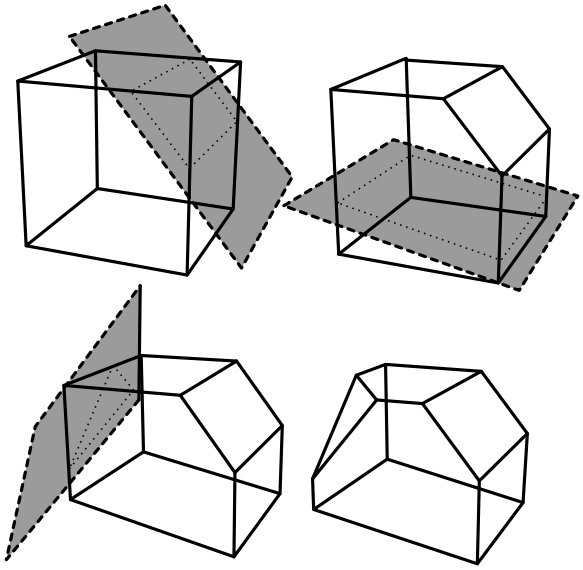
Obrázek 5.4: Funkce tangens pro transformaci z uniformního rozložení pravděpodobnosti pro koncentraci bodů okolo dané souřadnice. Rovnice funkce je $y = a \cdot tan(dx - b) + c$, kde $d = atan(\frac{1-c}{a}) + b$ a $b = -atan(\frac{-c}{a})$. Parametr a lze odvodit například ze síly nárazu při kolizi a ve výsledku ovlivní rozptyl bodů. Parametr c je odvozen z normalizovaných souřadnic místa největší hustoty bodů v rámci obalového tělesa modelu, v rámci jedné ze souřadných os. Parametr c posouvá funkci tak, aby plytká část grafu byla v místě požadované nejvyšší hustoty. Parametry v obrázku jsou následující: a)$a = 0.1, c = 0.5$, b)$a = 1, c = 0.5$ c)$a = 0.1, c = 1$.



Obrázek 5.5: Při generování bodu uvnitř konvexního modelu je nejprve náhodně zvolen jeden trojúhelník modelu (náhodně vygenerovaný index). V trojúhelníku je zvolena strana ab, kde je náhodně vygenerováno posunutí po této úsečce a tak zvolen bod $r1$. Stejný postup je aplikován na úsečku $cr1$ a následně na $sr2$, kde s je střed obalové krychle modelu. Bod $r3$, který takto vznikne je hledaný výsledek. Pro neuniformní rozložení je možné vygenerovat 8 kandidátních bodů uvnitř modelu a podle zadané pravděpodobnosti vybrat nejbližšího k danému bodu. (Pokud bereme v úvahu 256 vláken na workgroup, je možné vygenerovat na jednom procesoru 32 bodů (256/8), kde výsledek pak je z těchto osmic vybrán v jednom warpu.)

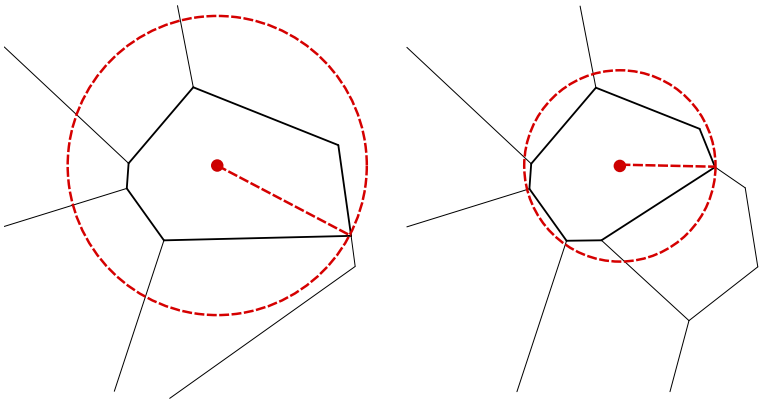
rovinu oddělující tyto dva body v polovině a touto rovinou usekne kus buňky. Dále pokračuje druhým nejbližším sousedem až do dosažení limitní vzdálenosti (Obrázek 5.6).

Každé vlákno ve skupině při osekávání nejprve testuje jeden vrchol úsečky (paralelně pro všechny vrcholy) a poté na základě těchto výsledků testuje každé vlákno jednu úsečku. V případě že úsečka leží za rovinou (vně buňky), je odstraněna. Pokud leží na druhé straně (uvnitř buňky), je ponechána a pokud protíná osekávací rovinu tak je rozpůlena. Takto vznikají nové vrcholy, které jsou po kontrole všech úseček spojeny do konvexní obálky a tvoří tak nové úsečky, ohraničující vzniklou díru v buňce.



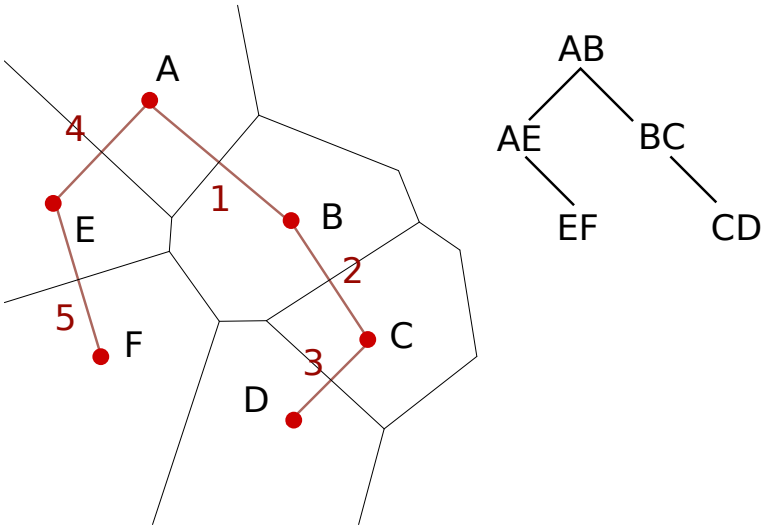
Obrázek 5.6: Tři iterace box cutting algoritmu. Začíná na AABB modelu a postupně tvaruje buňku osekávacími rovinami, ležícími mezi středem aktuální buňky a jeho nejbližšími sousedy.

Algoritmus končí v případě, že nová osekávací rovina leží za poloměrem tzv. maximální povolené vzdálenosti. Maximální povolenou vzdálenost je nutné přepočítat po každém schválení nové roviny 5.7. Tato vzdálenost je vypočítána jako vzdálenost od středu buňky k jejímu aktuálně nejvzdálenějšímu vrcholu.



Obrázek 5.7: Zmenšování maximální povolené vzdálenosti s narůstajícími stěnami. Jedná se o největší vzdálenost od středu buňky k jednomu z vrcholů buňky.

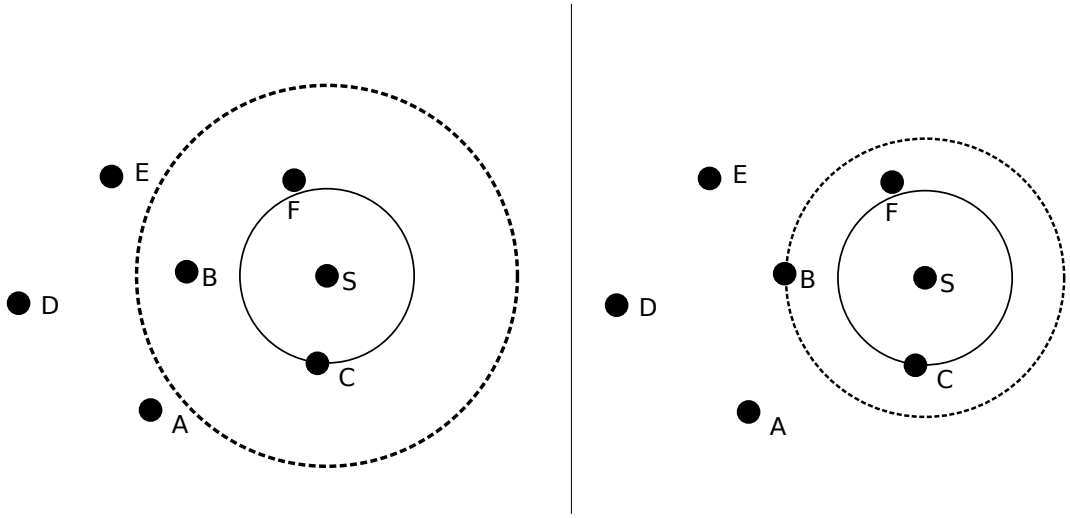
K optimalizaci hledání nejbližších sousedů lze použít binárního stromu [3]. Každý uzel tohoto stromu reprezentuje dva středy buněk. Do stromu jsou iterativně vkládány nové body tak, že je-li nově vkládaný střed blíže prvnímu středu v uzlu, je vložen do levého podstromu a naopak. Pokud podstrom není, je vložen nový uzel tvořící dvojici nově vkládaného středu a bližšího středu z rodičovského uzlu. Aby bylo možné použít tuto strukturu, je třeba vypočítat vzdálenost inicializačního bodu (bod jehož vložení způsobilo vznik uzlu) v daných uzlech k nejvzdálenějším potomkům. Část diagramu a odpovídající strom je znázorněn na obrázku 5.8.



Obrázek 5.8: Binární strom k nalezení nejbližších středů - čísla určují pořadí vkládání. Nový bod je vložen na stranu k sobě bližšího bodu daného uzlu.

Samotné hledání v tomto stromu pak probíhá pomocí zmenšování vyhledávací koule. Začíná se prohledávat od kořene a hledají se takové uzly, jejichž obalová koule (střed v iniciačním bodě a poloměr je vzdálenost k nejvzdálenějšímu potomku) koliduje s koulí vyhledávací. Vyhledávací koule má střed v aktuálním bodě, ke kterému hledáme sousedy a poloměr má na počátku maximální povolenou vzdálenost. Pokud při kolizích nalezneme bod, který je blíže středu vyhledávací koule tak její poloměr zmenšíme na vzdálenost k tomuto bodu od středu koule a tento bod uložíme jako možný výsledek. Jako omezení pro toto zmenšování existuje ještě vnitřní limit, což je poloměr od středu vyhledávací koule o velikosti vzdálenosti k naposledy nalezenému sousedu. Pokud by další bod zmenšil kouli pod tento limit tak ke zmenšení nedojde a jako nejlepší kandidát zůstane předešlý bod, který inicioval zmenšení. Příklad je na obrázku 5.9

Výhodou tohoto algoritmu je lepší využití GPU oproti Delaunay metodám, které výkon GPU zabírají postupně se zvyšujícím se počtem nových tetrahedronů. Oproti jiným algoritmům také BCA má na výstupu buňky v podrobné geometrické reprezentaci (vrcholy, hrany, strany), které lze jednoduše triangulovat a proto je tento algoritmus vhodný pro aplikace simulační povahy, kdy je nutné získat buňku jako 3D model.



Obrázek 5.9: Ukázka vyhledvání v optimalizačním stromu. Přerušovaný kruh je poloměr vyhledávací kružnice na počátku inicializován na maximální povolenou vzdálenost z BCA. Při nalezení bližšího bodu je tento poloměr patřičně zmenšen. Vnitřní kružnice je limitní a zabraňuje dalšímu zmenšení vyhledávací kružnice. Ve stromové struktuře tedy algoritmus nalezl nejprve bod B na který zmenšil vyhledávací kružnici. V předchozím hledání souseda byl nalezen bod C, který nyní definuje limitní poloměr vnitřní kružnice. V dalším kroku aktuálního hledání by byl nalezen bod F, kde by hledání skončilo.

5.4 Generování úlomků

Voroného diagram je tedy definován buňkami, kdy každá buňka je reprezentována množinou rovin. Nejjednodušším způsobem, jak získat úlomek objektu je osekávat původní objekt postupně rovinami buňky. Zde je popis naivního algoritmu:

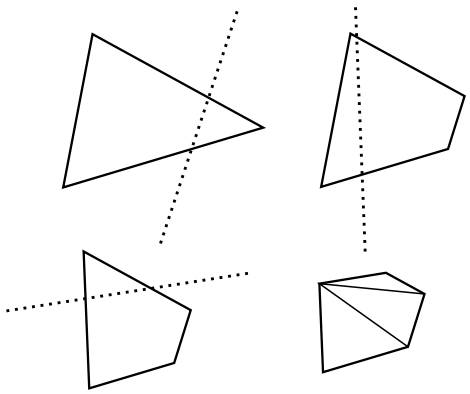
Paralelně pro každou buňku:
Pro každou rovinu dané buňky:
Pro každý trojúhelník objektu:

1. Zjistit vztah roviny a trojúhelníku - orientace všech tří bodů trojúhelníku vůči rovině
   - Pokud je trojúhelník na přední straně roviny, zahodit (je mimo buňku)
   - Pokud je trojúhelník na zadní straně roviny, uložit jako trojúhelník nového objektu
   - Jinak je trojúhelník protnut rovinou - pokud je jeden bod na zadní straně roviny, vytvořit nový trojúhelník s daným bodem a novými body v místech průsečíků hran spojených s daným bodem, jinak jsou dva body na zadní straně, tudíž podobným způsobem vytvořit dva nové trojúhelníky
2. Pokud došlo k dělení trojúhelníku, uložit nově vzniklé body

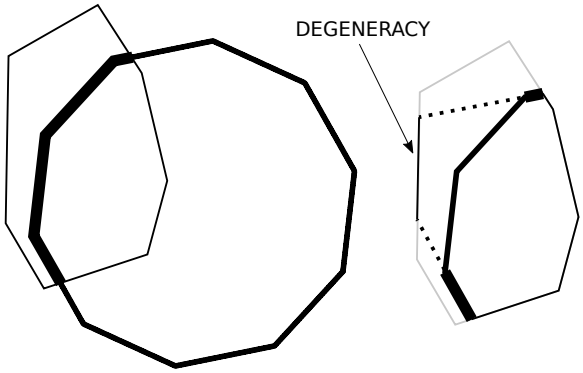
Triangulovat díru definovanou nově vzniklými body
Uložit nový objekt namísto původního a aplikovat další rovinu

Tento algoritmus však plně nevyužívá paralelismu poskytovaného GPU a vyžaduje mnoho přístupů do globální paměti, což může mít kritický vliv na výkon celé aplikace. Výhodnějším přístupem je rozdělit úlohu na menší podčásti.

V lokální paměti může být alokována bitová mapa, kde každý bit reprezentuje jeden trojúhelník modelu. Každé vlákno pak zkontroluje jeden trojúhelník vůči rovinám dané Voroného buňky. Jakmile je nalezena rovina, která trojúhelník vylučuje (všechny 3 body trojúhelníku leží vně), je nastaveno místo trojúhelníku v mapě na 0. Pokud testy projde, je nastaveno na 1. Speciální případ nastává, když je trojúhelník půlen rovinou. Takový trojúhelník je osekáván (možnost více rovin, které jej protínají jako na obrázku 5.10) podobně jako u BCA výše. Rovina, definována takovým trojúhelníkem je pak přidána do BCA buňky (označena jako netriangulovatelná) a buňka je zarovnána podle povrchu modelu. Celý proces naznačuje obrázek 5.11.

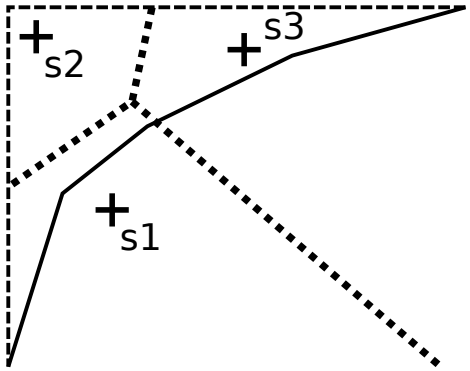


Obrázek 5.10: Jakmile protíná rovina trojúhelník povrchu, je třeba určit výsledný tvar, který vznikne z tohoto trojúhelníku a bude součástí povrchu úlomku modelu. Postup je podobný jako u BCA. Je udržováno pole úseček, na počátku jsou to tři úsečky reprezentující trojúhelník. Další roviny pak mohou dělit tento polygon, přičemž vzniká vždy jedna nová úsečka (stejně jako 2D polygon, protnutý přímkou). Na konci je polygon triangulován spojením jednoho bodu s úsečkami, které jej neobsahují.



Obrázek 5.11: Integrace povrchu modelu do úlomku, reprezentovaném Voroného buňkou (2D pohled). Silné čáry definují původní povrch modelu, tenké čáry Voroného buňku, přerušované čáry jsou roviny rozdělených trojúhelníků.

Na obrázku 5.11 je patrná jedna nevalidní stěna, která je vně povrchu modelu. K tomuto stavu dojde, pokud existují středy buněk vně povrchu modelu. Tak mohou vzniknout nechtěné stavy (Obrázek 5.12) které mohou být řešeny lokálně (posunem středu a přepočtem souřadnic bodů buňky) ale také mohou být eliminovány vhodným generováním středů (Obrázek 5.5). Původní roviny AABB, se kterými začíná výpočet jsou na počátku označeny jako netriangulovatelné.



Obrázek 5.12: Chyby způsobené vygenerováním středů buněk vně modelu. Silná čára reprezentuje geometrii modelu, přerušovaná tenká čára je AABB a tečkovaná silná čára je hranice mezi Voroného buňkami. $s1$ je validní buňka ale buňka definovaná středem $s2$ leží vně modelu, což by vyžadovalo další testy pro eliminaci buňky (tak by však nebylo možné přesně definovat předem počet buněk). Buňka $s3$ sice má průnik s povrchem, ale její střed leží mimo, což by znemožnilo pracovat v lokálním souřadném systému buňky a byla by nutná konverze po detekování tohoto stavu.

Na konci jsou podle bitové mapy zkopírovány validní trojúhelníky, uloženy vzniklé trojúhelníky z okrajových, které byly půleny rovinami buňky a samotné roviny buňky jsou triangulovány, mimo roviny označené jako netriangulovatelné.

5.5 Reakce na kolize

Není třeba testovat kolize statických objektů, které nikdy nenastanou. Proto při každé aktualizaci informací o objektu lze nastavit jeho stav v bitovém poli (pohybující se/statický). Obalová tělesa pohybujících se objektů jsou pak paralelně testovány s kolizemi obalových těles v BVH rekurzivně od kořene. Obalové těleso objektu může být rozšířeno ve směru pohybu či algoritmus detekce může brát v úvahu i změnu pozice v čase během daného simulačního kroku. Každé vlákno si tak udržuje seznam těles, se kterými dané těleso potenciálně koliduje. Kombinací těchto seznamů na globální úrovni lze předejít duplikátům při jemnějším výpočtu kolizí v narrow-phase. Po výpočtu-narrow phase kolizí lze sestavit seznam či matici jednotlivých přírůstků hybnosti pro každé těleso. (při kolizí určitou hybnost předá dalšímu tělesu, ale také určitou část může přijmout). Suma těchto hybností pro každé těleso vrací výslednou hybnost, ze které lze vypočítat výsledné rychlosti. Pro rotační pohyb je definována veličina moment hybnosti. Následující rovnice popisují obě veličiny:

$$\vec{p} = m\vec{v}; L = \vec{r} \times \vec{p} \tag{5.2}$$

kde

- $\vec{p}$ - hybnost
- $\vec{v}$ - rychlost
- m - hmotnost
- L - moment hybnosti
- r - průvodič (nejkratší vektor do místa kolize z osy otáčení)

Kapitola 6

Implementační detaily

Práce je implementována v jazyce C++ za využití grafického API OpenGL a knihovny GLFW. Je využito moderních technik, nabízených standardem C++17 a OpenGL verze 4.6 včetně DSA přístupu. Překlad je implementován pomocí nástroje Cmake. Pro jednoduchou práci s vektory a maticemi je dále využito knihovny GLM. Při programování je dbáno na objektově orientovaného paradigma a princip RAII.

6.1 Struktura aplikace

Aplikace se skládá z následujících modulů:

- Asset Manager - zajišťuje načítání modelů, textur a dalších potřebných dat, tato data také uchovává a zpřístupňuje pomocí konstantních ukazatelů
- Camera - zajišťuje pohyb a nastavení kamery
- Entity - obecná třída pro veškeré elementy scény, nejdůležitější třída, která dědí z Entity je třída Object zastupující fyzikální tělesa v simulaci
- GPU Manager - zajišťuje komunikaci s GPU, inicializaci OpenGL, vykreslování a spouštění kernelů
- Window - vytváří okno a propojuje jej s OpenGL, zachytává uživatelské vstupy
- Scene - modul, propojující všechny výše zmíněné, vytvoří scénu, rozmístí objekty a zajišťuje logiku simulace včetně zpracování vstupů
- Main - vstupní bod, hlavní smyčka simulace a inicializace modulů

Aby bylo možno využít GPU pro výpočty fyziky atd. je využito konceptu compute shaderů. Tyto shadery leží mimo vykreslovací grafickou pipeline, avšak mají přístup ke všem bufferům a uniformním proměnným, přístupným ostatním shaderům. Každý kernel je nad compute shaderem spuštěn samostatně, přičemž mezi sebou mají umístěny bariéry zajišťující globální synchronizaci. Výpočty compute shaderu musí předcházet samotné vykreslování, aby byla scéna vykreslena v aktualizovaném stavu. V compute shaderech jsou spouštěny kernely, zajišťující výše popsané výpočty na GPU. Mezi kernely, spouštěné v každém snímku patří:

- updateObjects - aktualizuje pozice objektů podle rychlostí, počítá aktuální AABB a následně Mortonův kód (MC) objektu
- radixSortMorton - používá radix sort algoritmus pro seřazení MC
- buildBVH - sestavuje BVH na základě seřazené sekvence MC
- collisions - vypočítá kolize na základě BVH a následnou reakci (v implementaci zjednodušeně, neuvažuje vícenásobné kolize)

V případě nutnosti tříštění objektu je spuštěna následující sekvence kernelů:

- voronoiPoints - generuje náhodně středy buněk
- voronoiPtTreeMark - přiřadí vstupní body do daných míst optimalizačního stromu
- voronoiPtTreeAlloc - alokuje nové uzly stromu (mark a alloc jsou spouštěny iterativně dokud jsou volné body)
- voronoiPtTreeDistances - počítá vzdálenost uzlu (od inicializačního bodu k nejvzdálenějšímu potomku)
- voronoi - samotné sestavení diagramu a dělení modelu

6.2 Reprezentace scény na GPU a vykreslování

V globální paměti GPU je alokováno několik bufferů pro práci s daty scény. Mimo klasické buffery pro reprezentaci modelů jako je vertex a element buffer je třeba také bufferu pro ukládání vykreslovacích příkazů, matic modelů, informací o objektech (fyzikální vlastnosti, rychlosti apod.) a další specifické buffery pro použití v daných kernelech jako je buffer pro obalová tělesa objektů, pro uložení stromové struktury bvh, pro mortonovy kódy a pomocný obecný buffer pro dočasná data. U těchto bufferů je nutno předem alokovat dostatek místa, která lze částečně odvodit od počtu a velikosti objektů scény. Mezi hlavní buffery patří:

- Vertex buffer {position: 3xfloat32, normal: 3xfloat32, texture coordinates: 2xfloat32} - vrcholy všech modelů scény
- Element buffer {1xuint32} - indexy pro všechny modely scény (indexed rendering)
- Draw commands {5xuint32} - draw command pro každý model (indirect rendering)
- Object information {21xfloat32, 3xuint32 (může se lišit podle požadavků simulace)} - vlastnosti objektů (position, velocity, mass...)
- Model matrices {4x4xfloat32} - použito ve vertex shaderu, transformační matice pro vrcholy, jedna pro každý model
- Bounding boxes {3x2xfloat32 (xyz bounds)} - pro urychlení výpočtů, jeden pro každý model
- General buffer - Volné místo, které může být znovu použito pro různé buffery (v našem případě např. Voronoi středy, stromové struktury, mortonovy kódy ...)

Pro snížení komunikace mezi GPU a CPU je využito nepřímého vykreslování tak, že jednotlivé vykreslovací příkazy jsou uloženy na GPU a celé vykreslení je spuštěno jedním voláním knihovní funkce OpenGL, přičemž grafická karta pak může pracovat pouze se svou pamětí bez účasti CPU. K ušetření místa je také použito instancování a indexace vrcholů. V případě instancí je geometrie modelu uložena v paměti jen jednou, přičemž jej lze do scény vykreslit vícenásobně s různými vlastnosti a transformacemi. U indexace vrcholů je využito element bufferu, kde jsou indexy vrcholů z vertex bufferu. Tak je eliminována nutnost ukládat často se vyskytující vrcholy v paměti vícekrát, například pro více trojúhelníků sdílející jeden vrchol. Další možností ušetření místa je kvantizace, kdy normálu a texturovací souřadnice je možné uložit do dvou 32 bitových hodnot (3 souřadnice normály po 10 bitech a 2 texturovací souřadnice po 16 bitech).

6.3 Použité algoritmy

Tato sekce obsahuje výčet některých postupů, které byly použity při implementaci. Jedná se o konkrétní doplnění částí návrhu, jejichž implementace nemusí být zcela jasná či nabízí více možných řešení.

Principy paralelního dělení prostoru

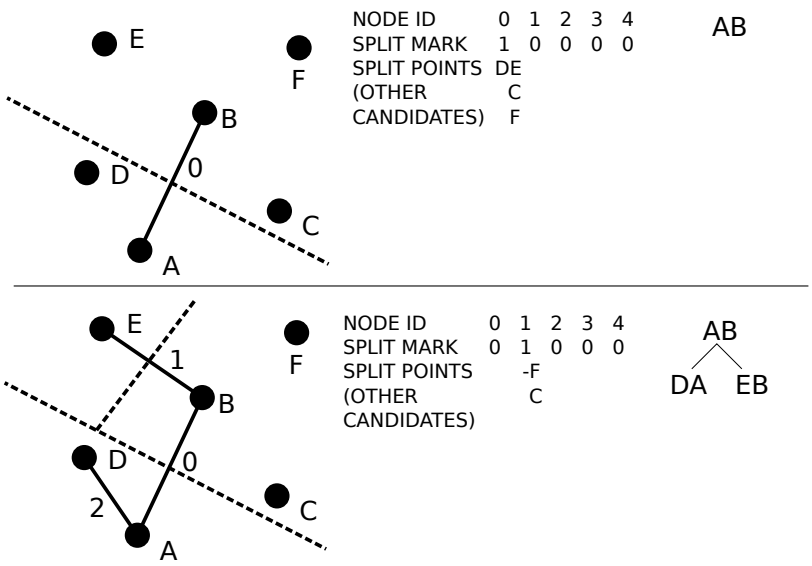
Zatímco BVH lze sestavit paralelně díky využití Mortonova kódu, pro optimalizační strom výběru nejbližších sousedů je nutno navrhnout paralelní verzi, původně sekvenčního algoritmu. Zejména je nutno vyřešit alespoň částečně paralelní stavbu stromu a alokaci uzlů. K tomu slouží dva kernely (mark a alloc). V prvním kernelu si každý bod ze vstupní množiny (ty které nejsou již zadané ve stromu) vybere, kam ve stromu patří. Na počátku je inicializován kořen se dvěma náhodnými body. Každý bod má také ve zvláštním bufferu uloženou cestu kam patřil naposledy a odtud bude v další fázi pokračovat, aby nemusel procházet celý strom znova. Kam bod patří je rozhodnuto tak, že je vybrán uzel z bufferu cesty (na počátku kořen) a bod pak může iniciovat štěpení tohoto uzlu buďto do pravého, nebo do levého potomka, podle toho zda je bod blíže pravému či levému bodu uzlu.

Uzly, které budou štěpeny jsou označeny v bitové masce, stejně tak je v dalším bufferu uložen index bodu, který bude daný uzel dělit. V případě, že více bodů bude dělit jeden uzel, je možné uložit výsledný index bodu pomocí operace atomického minima při požadavku determinismu. Pokud není náhodnost překážkou tak lze nechat vlákna všechna zapsat na dané místo v bufferu indexů bodů (kde každý uzel má dvě hodnoty, levé a pravé dělení) a jedna hodnota nakonec na tomto místě zůstane (Obrázek 6.1).

Ve druhé fázi stačí spustit 2^{level} vláken pro vytvoření nové úrovně uzlu. Každé vlákno se namapuje na štěpené uzly, poznačené v bufferech výše. Tak jsou alokovány nové uzly a body, které iniciovaly jejich vznik jsou přesunuty mimo vstupní množinu záměnou s posledními, nebo je lze poznačit jako neaktivní v přídavné bitové masce.

Urychlení hledání nejbližších sousedů

Implementací paralelní redukce, došlo k výraznému urychlení jak ukazuje graf 7.2 i oproti optimalizačnímu stromu. Prvním krokem je minimalizace přístupů do globální paměti při průchodu vstupních bodů. Do lokální paměti jsou při inicializaci buňky vypočítány vzdálenosti od aktuálního středu buňky ke všem ostatním bodům. Při hledání n-tého nejbližšího souseda je použito druhé pole s indexy těchto bodů, na stejných pozicích jako jsou dané



Obrázek 6.1: Paralelní stavba optimalizačního stromu bodů. Nalevo lze vidět spojování bodů do uzlů, které rozpůlí daný prostor. Napravo je znázorněna tabulka bufferů, kde je označeno které uzly budou děleny na aktuální úrovni a které body do nich budou vloženy.

vzdálenosti. Samotné hledání je inspirováno zmenšováním vyhledávací koule u optimalizačního stromu. Do nového pole jsou uloženy jen ty vzdálenosti (a asociované indexy), které jsou větší než poslední nalezená vzdálenost a menší než maximální povolená vzdálenost. Nad tímto polem vzdáleností možných sousedů je provedena paralelní redukce kdy operátorem, který je vykonáván nad daty je přepis první hodnoty druhou, pokud je druhá hodnota menší (spolu se vzdáleností je přepsán i asociovaný index). Tak je z možných vzdáleností sousedů nalezena minimální, která je výsledkem. Pokud nejsou žádné vzdálenosti vybrány pro redukci, proces dělení buňky končí.

Operace při dělení buňky

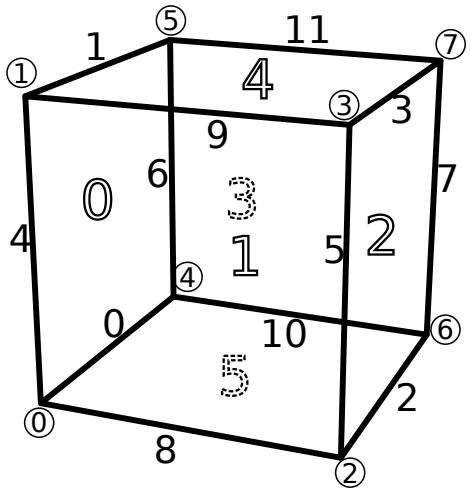
V této podsekci jsou uvedeny konkrétní implementační postupy pro určité operace, nutné během dělení buněk v BCA.

Náhodná čísla Pro generování náhodných souřadnic vstupních středů buněk je nutno implementovat vlastní generátor, jelikož GLSL nenabízí nativní funkci pro generování náhodných čísel. Výpis 6.1 popisuje možný pseudonáhodný generátor uniformního rozložení v intervalu $< 0.0, 1.0 >$. Další možností by bylo nechat vygenerovat texturu či pole hodnot na CPU a na GPU pouze číst tyto hodnoty s daným posunutím. Jako seed jsou zvolena čísla odvozená násobením ID vlákna a počtu bodů, dále násobené souřadnicemi vybraných trojúehlníků modelu.

A~= 12.9898;
B = 78.233;
C = 43758.5453;
fract(sin(dot(coord.xy ,vec2(A,B))) ∗ C);

Výpis 6.1: Pseudo-náhodný generátor čísel uniformního rozložení pro GPU [12]

Reprezentace buňky a inicializace AABB V předchozí sekci byl popsán způsob reprezentace buňky pomocí úseček, který je zde upřesněn na příkladu. Zároveň je zde popsán způsob paralelního vygenerování obalové krychle (počátek BCA) tak, aby odpovídala této reprezentaci, zejména v oblasti vztahů a odkazů mezi úsečkami, rovinami a vrcholy. Krychle je popsána na obrázku 6.2.



Obrázek 6.2: Indexace hran, vrcholů a stran krychle, která definuje tvar buňky na počátku BCA. Dvojitou čarou jsou označeny čísla stran, v kroužku jsou čísla vrcholů a obyčejná čísla označují hrany.

Při znalosti rozsahu AABB modelu (vektor *aabb* obsahující minimum a maximum obalové krychle jako dva prvky pro každou osu z xyz) lze vygenerovat paralelně celou krychli. Následující vztahy byly odvozeny rozepsáním indexů do tabulek a nalezením platné rovnice pro mapování lokálního ID vlákna na dané indexy. Vrcholy lze vygenerovat pomocí rovnice:

$$vertex[lid] = \{aabb.x[(lid/2)\%2]; aabb.y[lid\%2]; aabb.z[lid/4]\} \tag{6.1}$$

Hrany podle rovnic:

Pro $lid < 4$:

$$edge[lid] = \{lid; 4 + lid; lid/2 * (lid + lid\%2); 5 - lid\%2 * lid\} \tag{6.2}$$

Pro $4 <= lid < 8$:

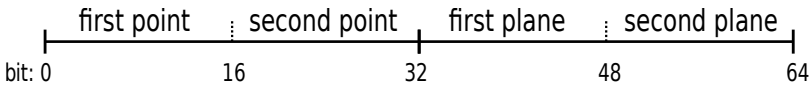
$$edge[lid] = \{(lid - 4) * 2; lid - (3 - lid\%4); (lid\%2) * (lid/2 - 1); lid - 3 - lid/7\} \tag{6.3}$$

Pro $8 <= lid < 12$:

$$edge[lid] = \{(lid/10) * 2 + lid\%4; (lid/10) * 2 + lid\%4 + 2; lid/2 - 3 + lid/10; 5 - lid\%2\} \tag{6.4}$$

kde

- *aabb* je vektor obsahující tři dvojice maxima a minima AABB na dané ose
- *lid* je ID vlákna v rámci pracovní skupiny
- *vertex*[] je pole vrcholů ve tvaru trojrozměrných vektorů souřadnic
- *edge*[] je pole hran ve tvaru {vrchol A; vrchol B, strana I, strana J} (Obrázek 6.3)



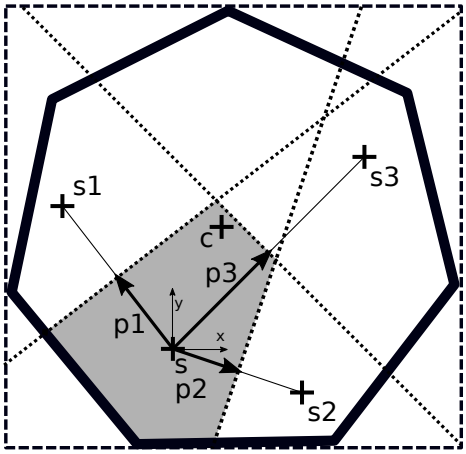
Obrázek 6.3: Reprezentace hrany jako uvec2 (2x32 unsigned int) v lokální paměti. Každá 16bit hodnota je index do pole vrcholů/rovin.

Připomínám že dělení/násobení n-tou mocninou 2 u typu *unsigned int* lze implementovat jako bitový posun doprava/doleva o n bitů a modulo podobně pomocí operace *and*, kde operandy jsou daná hodnota a mocnina 2 snížena o 1.

Implementace BCA Algoritmus 1 popisuje detailně hlavní smyčku BCA postupu.

Štěpení buňky a lokální souřadný systém Veškeré souřadnice buňky a také souřadnice trojúhelníků modelu jsou zpracovány v lokálním souřadném systému buňky jako je naznačeno na obrázku 6.4. Výhodou je možnost reprezentovat ořezovou rovinu jako jeden tříprvkový vektor, kde jeho směr udává orientaci roviny a velikost její pozici, jak popisuje rovnice:

$$\vec{p} = \{x, y, z\}, plane = \{\vec{p}, \frac{\vec{p}}{|\vec{p}|}\} \tag{6.5}$$



Obrázek 6.4: Objekt (zvýrazněný polygon) v obalovém tělese (přerušovaný čtverec). Obrázek znázorňuje roviny Voroného buňky (šedá oblast), kdy rovina je reprezentována jako vektor *pn*, směřující ven ze středu buňky, přičemž *s* značí aktuální střed buňky a *sn* středy n sousedících buněk. Veškeré souřadnice jsou v lokálním souřadném systému buňky s počátkem v bodě *s*, zatímco střed lokálního systému modelu je v bodě *c* (zpravidla střed geometrie). Přerušované čáry znázorňují hranice buňky.

Algorithm 1: BCA pseudokód

```
Data: množina vygenerovaných středů, AABB modelu
Result: jedna Voroného buňka (její hrany, vrchoy a stěny)
site = getCellCenter(WorkGroupId);
initBoundingBoxCell();
avaliablePlanes = true;
while avaliablePlanes do
    maxDist = cellFurthestPoint();
    neighbor = nextNeighbour(site,maxDist);
    if neighbor != null then
        plane = (neighbor-site)/2.0;
        checkCellVerticesParallel(plane);
        splitEdgesParallel();
        memoryClear();
        createHullEdgesParallel();
    else
        avaliablePlanes = false;
    end
end
```

kde

- *getCellCenter()* vrací střed pro aktuálně zpracovávanou buňku (podle ID pracovní skupiny)
- *initBoundingBoxCell()* inicializuje buňku na AABB modelu
- *cellFurthestPoint()* vrací nejvzdálenější bod aktuální geometrie buňky od středu (lze použít paralelní redukce)
- *checkCellVerticesParallel()* nastaví v bitové masce vrcholů jejich orientaci vůči aktuální osekávací rovině
- *splitEdgesParallel()* provede půlení/odstranění konfliktních hran buňky
- *memoryClear()* odstraní nevalidní vrcholy/hrany, aby nedošlo k přetečení v lokální paměti (lze využít prefixové sumy či atomického počítadla, validní elementy získají nové indexy do svých polí a přepíší tak staré hodnoty)
- *createHullEdgesParallel()* vytvoří nové hrany ze vzniklých vrcholů, které ohraničí díru vzniklou po oseknutí (není třeba používat geometrické algoritmy pro konvexní obálky, ale stačí uložit si, které strany buňky byly napojeny na půlenou úsečku ze které vznikl nový bod, v hranu pak jsou spojeny body, sdílející stejný index strany)

Orientace vrcholu lze otestovat jako:

$$normalize(\vec{p}) \cdot (\vec{v} - \vec{p}) > 0 \tag{6.6}$$

kde

- p je vektor reprezentující rovinu

- v je vrchol

Nový vrchol při dělení hrany získáme pomocí:

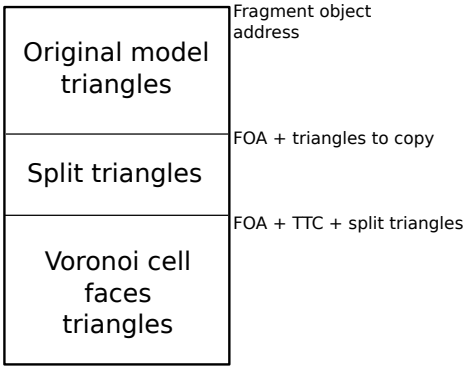
$$\vec{v} = \vec{a} + \frac{normalize(\vec{p}) \cdot (\vec{p} - \vec{a})}{normalize(\vec{p}) \cdot (\vec{b} - \vec{a})}(\vec{b} - \vec{a}) \tag{6.7}$$

kde

- a, b jsou vrcholy původní hrany
- p je vektor reprezentující rovinu
- v je výsledný vrchol

Ukládání úlomků

Po dokončení tříštění musí každá pracovní skupina uložit svůj úlomek (který se následně chová stejně jako ostatní objekty simulace) do globální paměti spolu se strukturou informací o objektu a strukturou draw command. Aby bylo možné paralelně ukládat úlomky, je nutné nejprve spočítat, kolik trojúhelníků bude zkopírováno z původního modelu, kolik jich bude půleno a kolik nových trojúhelníků tak vznikne a kolik trojúhelníků vznikne triangulací stran Voroného buňky. Tento počet pak udává celkovou velikost úlomku a je atomicky přičten ke globálnímu počítadlu vrcholů, jehož originální hodnota bude sloužit jako počáteční index pro ukládání geometrie úlomku. Paměť pro jeden úlomek vypadá jako na obrázku 6.5.

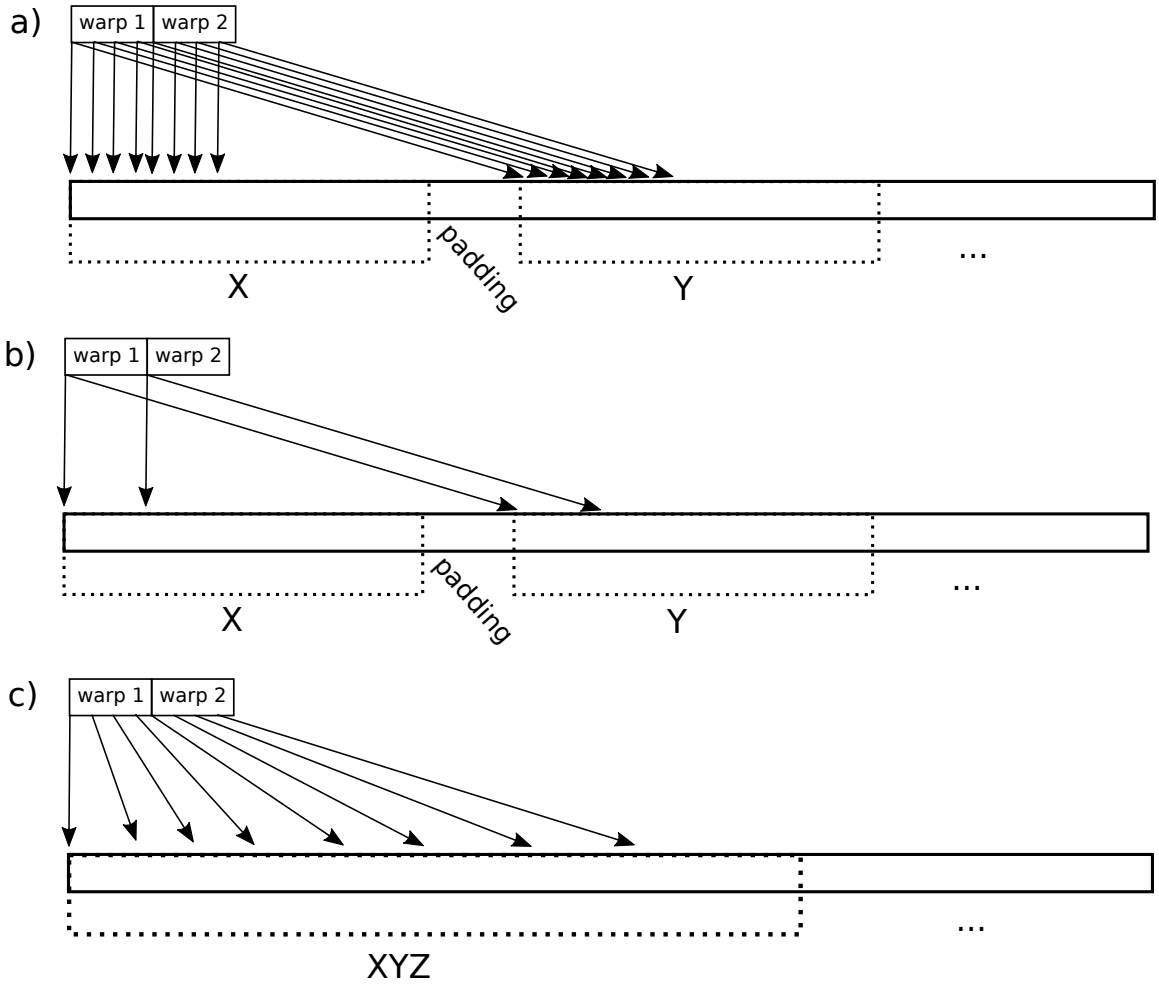


Obrázek 6.5: Rozvržení paměti pro úlomek ve vertex a element buffferu.

Zarovnání paměti

GPU čte z globální paměti po blocích (32/128bit), které jsou cachovány a mohou být využity pro více operací. Samotná latence při vykonávání paměťových operací je překrývána vykonáváním dalších warpů. Tudíž je výhodné zarovnat data v paměti a následně je číst zarovnaně i za cenu dodatečného volného místa, které vyplňuje zbylý prostor. OpenGL zarovnává interně např. *vec3* na *vec4*, přičemž v shaderech pracujeme s *vec3*. Například vstupní body Voroného diagramu byly nejprve uloženy jako *float[3]* (bez zarovnání), aby bylo možné jednoduše paralelně generovat body po souřadnicích. Při změně postupu, generování celých bodů v jednom vlákně a změně uložení na *vec3* došlo ke znatelnému zrychlení přístupu k tomuto poli (zrychlení okolo 100ms při 300 bodech). K podobnému urychlení

došlo při změně dat ve struktuře uzlů optimalizačního stromu (struct se třemi 32bit hodnotami byl zarovnán na 4). Stejně tak lze urychlit čtení vrcholů při dělení modelu tak, že jsou souřadnice uloženy prokládaně ve formátu *xxxx...yyyy...zzzz...* (tedy první souřadnice všech vrcholů za sebou, druhé atd...) a čtení nebude prováděno po celých vrcholech, ale zvlášť po souřadnicích. Obrázek 6.6 popisuje tento rozdíl.



Obrázek 6.6: Dva typy zarovnání vertex bufferu a následné čtení. a) Čtení prokládaných souřadnic, kdy warp čte blízké souřadnice. Tato situace je pak optimalizována díky čtení bloků na b). c) Klasické čtení po strukturách vrcholů, obsahujících trojice souřadnice. Vzniká mnoho různých čtení po celé délce bufferu.

Kapitola 7

Měření

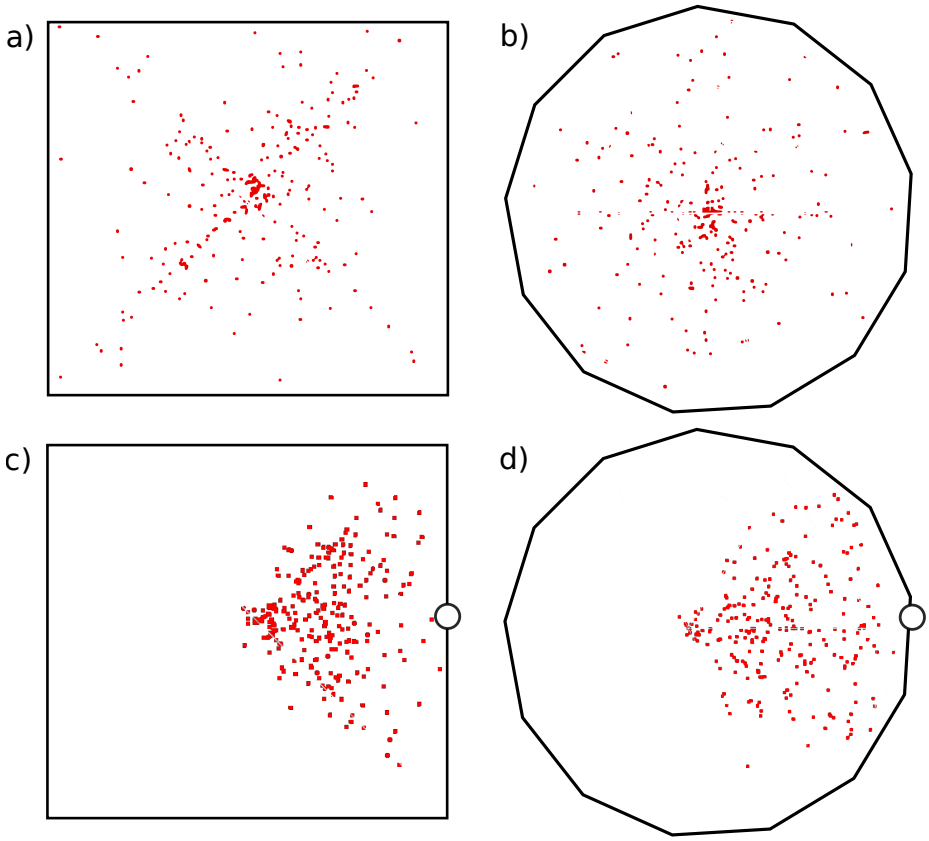
Následuje několik měření výkonnosti aktuální verze implementace výše popsaných postupů, zejména BCA techniky a samotného dělení objektů. Měřená aplikace neobsahuje všechny optimalizace, které by byly nutné pro maximální výkon implementovaných technik. Další optimalizace, zejména spojené s technikami efektivního GPU programování by mohly přinést znatelné urychlení. Měření však může nastínit alespoň základní rozdělení práce při výpočtech, možnost využití výše popsaných algoritmů a problémová místa implementace. Samotnému výkonnostnímu měření předchází zhodnocení kvality generátoru náhodných čísel. Tabulka 7.1 poskytuje informace o prostředí, kde bylo měření prováděno.

CPU	Intel Core i3-2100, 3.10GHz
RAM	12GB
GPU	Nvidia GeForce GTX 550 Ti
OS	Arch Linux 4.16.3-1 64bit
GPU Driver	Nvidia Driver 390.48

Tabulka 7.1: PC sestava, na které byla prováděna implementace a měření

7.1 Náhodné body

Rozložení náhodných středů Voroného buněk může značně ovlivnit kvalitu výsledku, proto je zde předložena ukázka vygenerovaných bodů na obrázku 7.1. Lze vidět úskalí použité techniky, kdy jsou pozice bodů závislé na trojúhelnících modelu a spojnici se středem. V místech rohů a hran je hustota bodů menší i při uniformním rozložení. V případě jednoduchého modelu lze navíc vidět větší koncentraci bodů na spojnicích středů trojúhelníků se středem modelu. Dále při neuniformním rozložení se body nechovají přesně podle očekávání, kdy bude shluk bodů okolo bodu kolize. Paradoxně však modeluje toto rozložení, kdy jsou body okolo bodu kolize více rozptýleny realitu lépe. Při nárazu objektů je energie většinou rozptýlena do okolí bodu kontaktu a tříštění se nejvíce projeví v určitém okruhu od bodu kontaktu na povrchu objektu. Pro potřebu simulace je tedy implementovaná metoda dostačující.

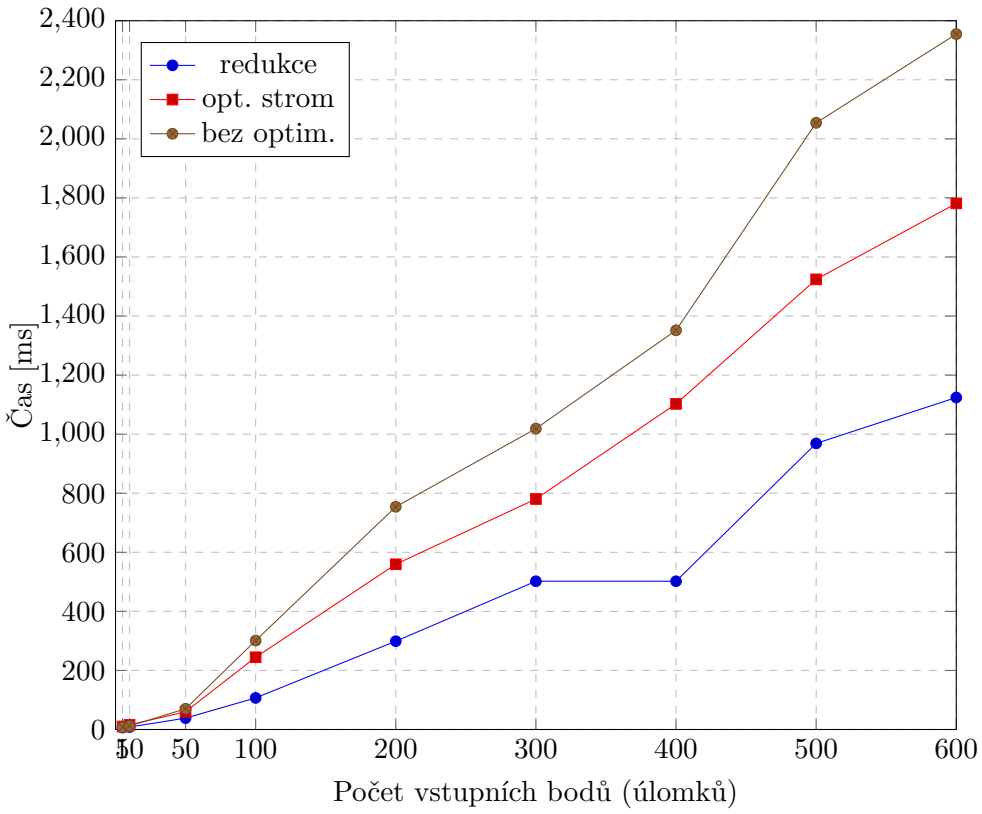


Obrázek 7.1: 500 náhodně vygenerovaných bodů pomocí metody náhodného vzorkování geometrie modelu. a) uniformní rozložení na modelu krychle, b) uniformní rozložení na modelu koule, c) větší hustota pravděpodobnosti okolo kontaktního bodu u krychle, d) stejné rozložení jako předchozí u koule. Modely s hustší triangulací a větší diverzitou orientací rovin trojúhelníků vracejí lepší výsledky než jednoduché tvary.

7.2 Rychlost výpočtů

Měření rychlosti odhalilo zajímavá fakta o implementaci, zvláště v oblasti vyhledávání nejbližšího souseda. Zatímco optimalizační strom přinesl urychlení výpočtů oproti brute-force přístupu hledání n-tého nejbližšího souseda, implementace paralelní redukce a omezení přístupů do globální paměti překonalo všechna ostatní řešení, jak zřetelně ukazují výsledky na grafu 7.2. Důvodem je pravděpodobně zarovnaný přístup do paměti (zatímco průchod stromem může požadovat čtení z různých míst pole uzlů), omezení přístupů do globální paměti a možnost masivní paralelizace, která u stromové struktury není zcela možná. Paralelní redukce však pracuje s lokální pamětí, která je při tříštění naplno využita. Tak vzniká omezení na 800 úlomků. V případě nutnosti jemnějšího tříštění by bylo namísto redukce použít právě tento strom, který by dále bylo možné optimalizovat částečnou paralelizací jeho průchodů, která již nebyla v tomto projektu implementována. Více úlomků by však bylo možno řešit také rekurzivním dělením vzniklých úlomků s dosavadní implementací dělení objektu.

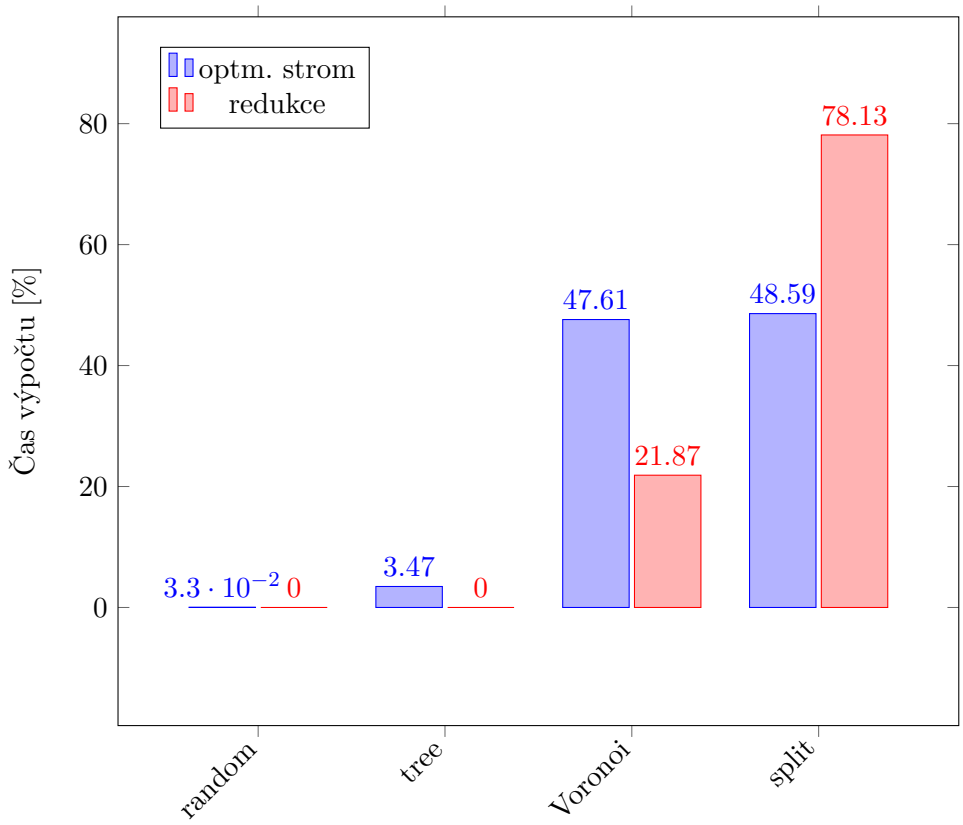
Na grafu 7.3 je znázorněno rozložení práce mezi kernely. Je patrné, že proces tříštění výrazně zpomaluje dělení geometrie modelu. Příčinou je velký počet přístupů do globální paměti, kde je uložena originální geometrie a následné zapisování dat nových úlomků. Proces



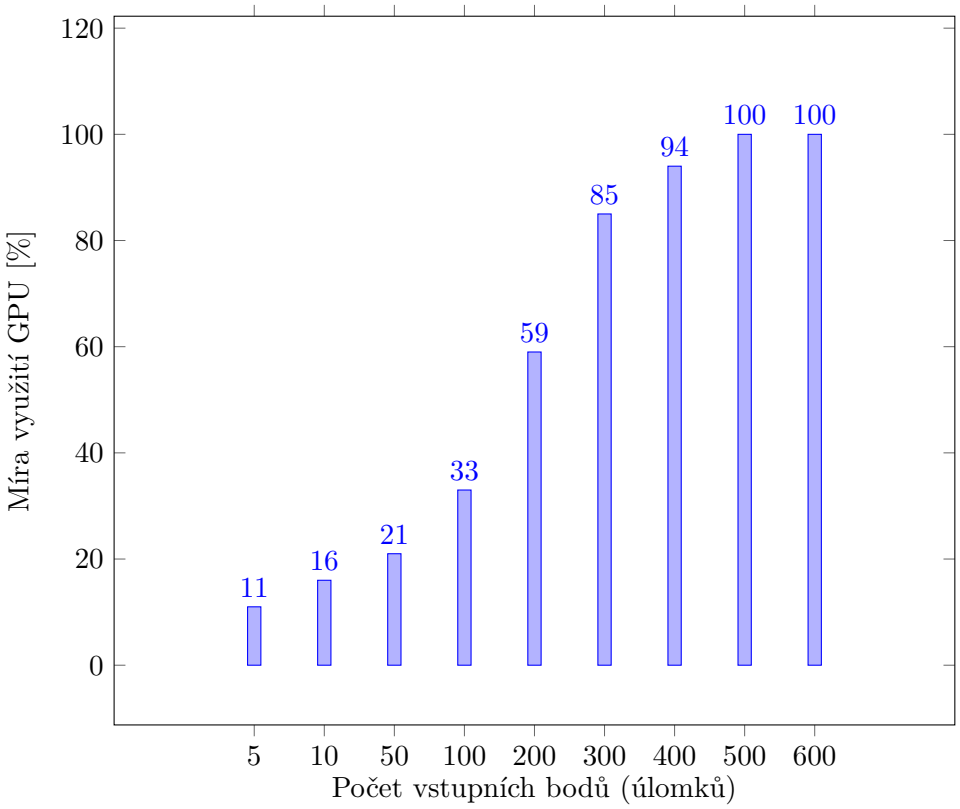
Obrázek 7.2: Naměřené hodnoty času běhů kernelů, zajišťujících štěpení modelu typu icosphere složeného ze 42 vrcholů. Body byly náhodně vygenerovány v uniformním rozložení pravděpodobnosti. Pozice bodů mají také vliv na výkon. Větší koncentrace bodů na jednom místě způsobuje mírné zpomalení.

čtení z paměti navíc zpomaluje indexované uložení vrcholů, znemožňující ideální zarovnaný přístup. Tato část výpočtu by byla dále vhodným kandidátem k dalším optimalizacím.

Graf 7.4 popisuje míru využití GPU při výpočtech. Důležitým prvkem návrhu GPU aplikace je právě vhodné rozdělení práce tak, aby bylo efektivně zaměstnáno co nejvíce vláken a nedocházelo ke zbytečnému čekání při čtení z paměti či využití jen zlomku alokovaných vláken v rámci pracovní skupiny. Tento bod implementovaná aplikace splňuje. Další vylepšení by tedy měla být zaměřena spíše na efektivitu kódu v rámci vláken.



Obrázek 7.3: Rozložení práce mezi jednotlivé kernely podle doby trvání výpočtů. V grafu jsou uvedena dvě rozložení, verze s optimalizačním stromem a s paralelní redukcí. Generování náhodných bodů je označeno jako *random*, stavba optimalizačního stromu jako *tree*, konstrukce Voroného diagramu jako *Voronoi* a dělení modelu, včetně uložení nových úlomků do paměti jako *split*.



Obrázek 7.4: Míra využití výpočetního výkonu GPU při tříštění modelu typu icosphere složeného ze 42 vrcholů. Měření probíhalo pomocí *query* přepínače u příkazu *nvidia-settings*.

Kapitola 8

Závěr

V rámci této práce byl navržen a implementován přímý postup generování Voroného diagramu pomocí BCA techniky a jeho aplikace při tříštění objektů na GPU. Tento postup byl zahrnut v tzv. simulaci křehkých těles, která spojuje samotné dělení geometrie s detekcí kolizí mezi objekty, výpočty fyziky a celkovou reprezentací 3D scény na GPU. Simulace křehkých těles je komplexní problém, umožňující aplikaci různých technik. Přesunutím těchto výpočtů na GPU je uvolněn výkon CPU pro další výpočty, spojené s hlavní aplikací. Primární motivací tohoto projektu bylo navrhnout řešení tak, aby bylo možné jeho užití například v počítačových hrách při běhu aplikace v reálném čase.

Výsledná implementace zcela nenaplňuje všechny původní požadavky na výkon, díky velkému množství metod a dalších optimalizací (zejména z oblasti paralelních návrhových vzorů a přístupů do paměti GPU), které by bylo třeba dále implementovat. Rychlost výpočtu při stovkách úlomků již není dostatečná, nicméně vizuální výstup a věrnost tříštění je na dostatečné úrovni, jak je možné nahlédnout v příloze A. Práce nicméně nabízí nové postupy a poukazuje na základě měření na nedostatky různých přístupů v poměrně málo zdokumentované oblasti práce s křehkými tělesy na GPU a byla zařazena do konference Excel@FIT 2018. V této oblasti výzkumu bych rád dále pokračoval, například v rámci své disertační práce při doktorském studiu.

Literatura

[1] Aurenhammer, F.: Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, ročník 23, č. 3, 1991: s. 345–405.

[2] Bergen, G. V. d.: A fast and robust GJK implementation for collision detection of convex objects. *Journal of graphics tools*, ročník 4, č. 2, 1999: s. 7–25.

[3] Blackpawn: Making Cellular Textures.
URL http://blackpawn.com/texts/cellular/default.html

[4] Cao, T.-T.; Nanjappa, A.; Gao, M.; aj.: A GPU accelerated algorithm for 3D Delaunay triangulation. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, 2014, s. 47–54.

[5] Cignoni, P.; Montani, C.; Scopigno, R.: DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed. *Computer-Aided Design*, ročník 30, č. 5, 1998: s. 333–341.

[6] Eberly, D.: Intersection of convex objects: The method of separating axes. *www. magic-software. com*, 2001.

[7] Harris, M.: Optimizing cuda. *SC07: High Performance Computing With CUDA*, 2007.

[8] Hecker, C.: Physics, part 4: The third dimension. *Game Developer*, 1997: s. 15–26.

[9] Jin, L.; Kim, D.; Mu, L.; aj.: A sweepline algorithm for Euclidean Voronoi diagram of circles. *Computer-Aided Design*, ročník 38, č. 3, 2006: s. 260–272.

[10] Ledoux, H.: Computing the 3d Voronoi diagram robustly: An easy explanation. In *Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on*, IEEE, 2007, s. 117–129.

[11] Lousada, P.; Costa, V.; Pereira, J. M.: Bandwidth and memory efficiency in real-time ray tracing. 2017.

[12] L'Ecuyer, P.; Simard, R.: A Software Library in ANSI C for Empirical Testing of Random Number Generators. Technická zpráva, Technical report, Département d'Informatique et de Recherche Opérationnelle Université de Montréal, 2002.

[13] Möller, T.: A fast triangle-triangle intersection test. *Journal of graphics tools*, ročník 2, č. 2, 1997: s. 25–30.

[14] Nanjappa, A.: *Delaunay Triangulation in R3 on the GPU*. Dizertační práce, 2012.

[15] Peringer, P.: Modelování a simulace. *Fakulta informačních technologií: Vysoké učení technické v Brně*, 2006: str. 206.

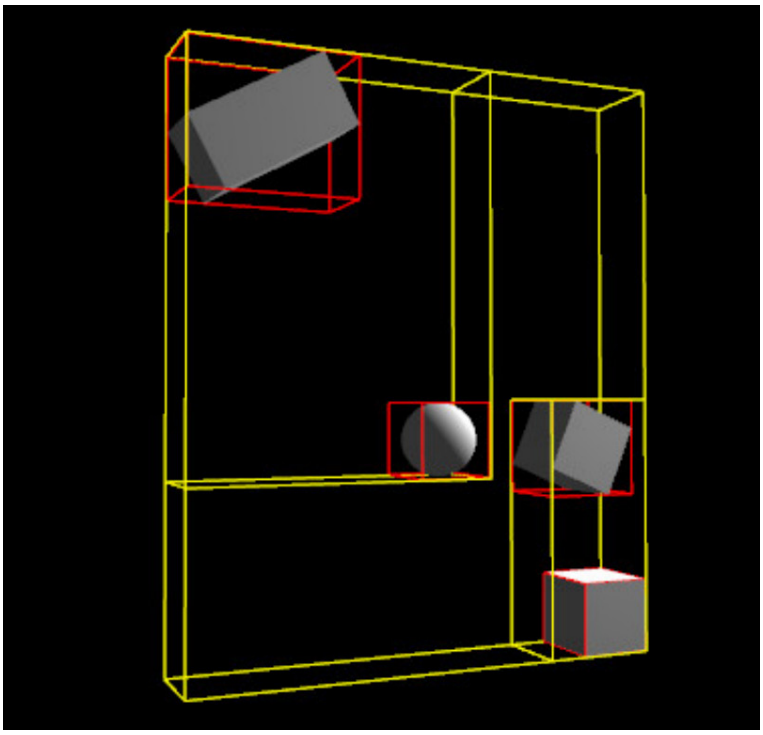
[16] Rebay, S.: Efficient unstructured mesh generation by means of Delaunay triangulation and Bowyer-Watson algorithm. *Journal of computational physics*, ročník 106, č. 1, 1993: s. 125–138.

[17] Tatourian, A.: Nvidia gpu architecture and cuda programming environment. *URL http://code. msdn. microsoft. com/windowsapps/NVIDIA-GPU-Architecture-45c11e6d*, 2013.

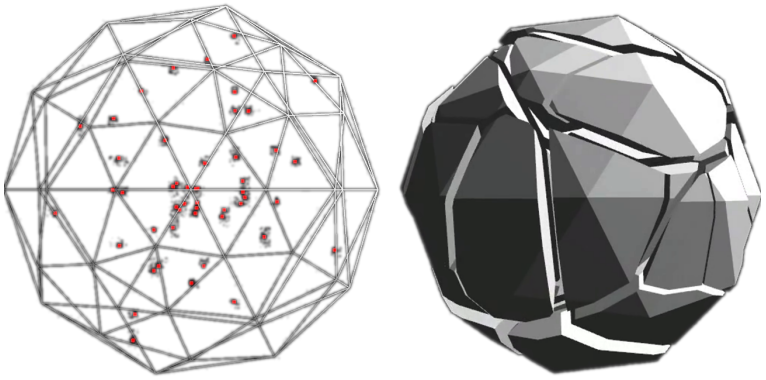
[18] Vasconcelos, C. N.; Sá, A.; Carvalho, P. C.; aj.: Lloyd's algorithm on GPU. In *International Symposium on Visual Computing*, Springer, 2008, s. 953–964.

Příloha A

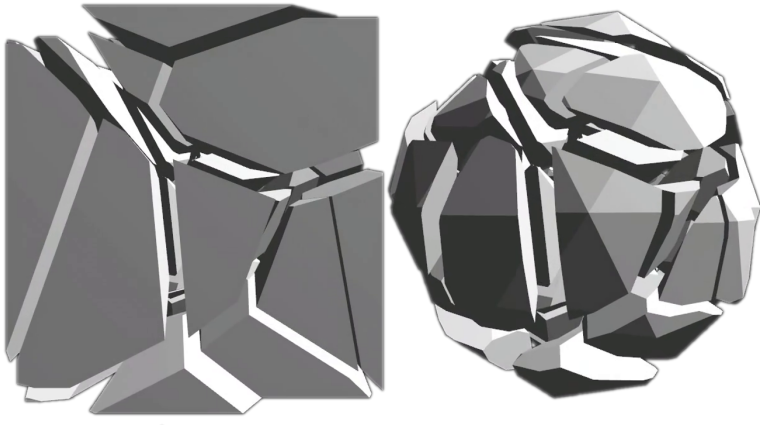
Obrázky z referenční aplikace



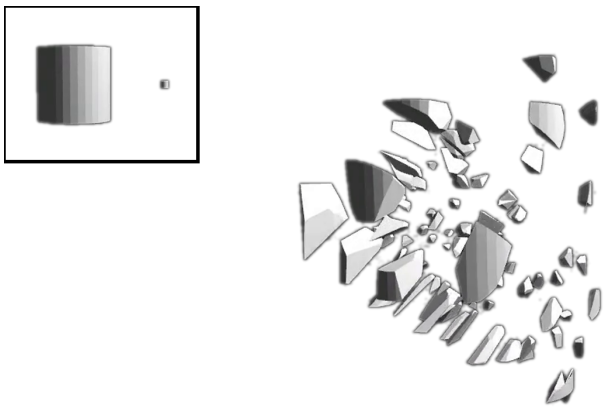
Obrázek A.1: Sestavení BVH na GPU nad jednoduchou scénou



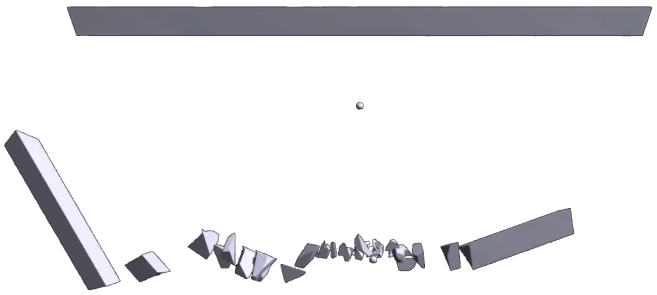
Obrázek A.2: Vlevo: kostra modelu s náhodně vygenerovanými vstupními body Voroného diagramu v uniformním rozložení. Vpravo: aplikace tříštění na základě vstupních bodů.



Obrázek A.3: Aplikace stejného Voroného diagramu na dva modely různé geometrie. Voronoi aplikovaný na krychli odpovídá přesné vizualizaci jeho buněk bez ohledu na geometrii modelu.



Obrázek A.4: Situace srážky objektů, kdy je velký válec roztříštěn malou kulkou. Více malých úlomků je generováno poblíž kontaktního bodu.



Obrázek A.5: Demonstrace rozložení úlomků při kolizi okolo kontaktního bodu většího objektu s menším.